

*Von der Carl-Friedrich-Gauß-Fakultät für Mathematik und Informatik
der Technischen Universität Braunschweig
genehmigte Dissertation zur Erlangung
des Grades eines Doktor-Ingenieurs (Dr.-Ing.)*

Generative Mesh Modeling

Sven Havemann

Institut für ComputerGraphik

Datum der Promotion 16. November 2005

1. Referent Prof. Dr. Dieter W. Fellner
Institut für ComputerGraphik
TU Braunschweig, Deutschland

2. Referent Prof. Dr. Heinrich Müller
Lehrstuhl für Informatik VII (Graphische Systeme)
Universität Dortmund, Deutschland

eingereicht am 15. Juli 2005

Preface

Ultimately unleash the incredible potential and the creative power of interactive three-dimensional computer graphics with 3D for the masses!

The great promises of 3D technology range from rapid prototyping and computer aided industrial design over intuitive graphical interfaces to far-fetched concepts such as virtual reality and cyberspace. We are entertained by interactive movies and games in cinema-like quality and our everyday life is fundamentally changed by innovative ways of presenting and selling goods and products in 3D over the internet. All this seems to be in range, if not already becoming a standard. And, of course, the whole industrial workflow is now completely digital and 3D, with high-tech companies collaboratively designing their future products by exchanging masses of highly sophisticated, compatible and inter-operable virtual prototypes every day.

Or is this only a myth?

The truth is in any case that 3D technology is not at all used as much as it could. Interestingly, this is not a matter of limited hardware resources, as today practically every newly sold computer has built-in support for interactive 3D. But still it is rarely the case that average computer users express their ideas about shape with the aid of a digital computer. – This thesis tries to analyze the reasons for this situation, and it identifies the *shape description problem* as one fundamental issue.

The initial spark of inspiration for a solution came from a very fruitful idea of Prof. Dieter Fellner, the *maintenance of semantic information*. This is a generally applicable concept with many incarnations, and it has stimulated much of the work in the computer graphics groups first at the university of Bonn, then in the newly founded institute of computer graphics at the technical university in Braunschweig.

Semantic information about shape is at the heart of generative modeling. A very concrete result of the presented research on shape descriptions is the *Generative Modeling Language*. It is even of practical use: The illustrations in the technical part of this thesis, 327 diagrams and 877 OpenGL screenshots, have been created exclusively using the GML. This thesis contains altogether 1616 little images, which may be a bit excessive. Some colleagues have pointed out that it can be read very much like a comic strip. This is probably true.

I am very much aware of, and grateful for, the powerful support from our group, from its past and present members. I am also particularly proud and grateful that Prof. Fellner has given me the long-time support and his encouragement to pursue one idea.

Diese Arbeit ist meiner Familie gewidmet – meiner Frau Miriam, deren stille Heiterkeit und milder Spott manche Zeile begleitete, und meinen Kindern Finn, Paula und Karl, die stets aufpassen daß ihr Vater mit beiden Beinen auf dem Boden bleibt. Danke für Eure Geduld!

Braunschweig, den 15. Juli 2005

Sven Havemann

Contents

Preface	iii
1 Introduction	1
1.1 The Shape Description Problem	1
1.2 Review of Applied Shape Design and Interactive Computer Graphics	7
1.2.1 Shape Acquisition	8
1.2.2 Shape Modeling with Procedural Modelers	9
1.2.3 Shape Modeling with Parametric CAD Software	11
1.2.4 Online Rendering: Interactive 3D	14
1.3 Modeling in Computer Graphics	18
1.3.1 Low-Level Shape Representations: Lists of Primitives	18
1.3.2 Shape Processing	20
1.3.3 Interactive Shape Design	21
1.3.4 Generative Shape Design	26
1.4 Short List of Difficult Problems with Current 3D Technology	31
1.5 Development of the Generative Mesh Modeling Approach	33
1.6 A New Type of 3D Technology: The Generative Modeling Language (GML)	50
1.6.1 Four-Layered GML Software Architecture and Thesis Overview	52
1.6.2 Features of GML-based 3D Technology	53
1.6.3 Potential: A Wealth of New Questions	55
2 Theoretical Foundation of Polygonal Meshes	59
2.1 Basic Facts of Algebraic Topology	59
2.2 Euler Operators	69
2.2.1 Euler Operator Example: Quadrangular Torus	72
2.2.2 Another way to build the Quad Torus	74
2.2.3 Properties of Euler Operators	75
2.2.4 Closedness and Completeness of Euler Operators	76
2.3 What is a Mesh? – A Definition	80
2.3.1 Mesh Definition	80
2.3.2 General Meshes in Computer Graphics: Indexed Face Sets	82
2.3.3 Are Meshes necessary for Rendering? – The OpenGL Answer	82
2.3.4 Mesh Manipulation with Euler Operators	85
2.3.5 A Note on Seemingly inconsistent Mesh Configurations	87
2.4 Hierarchical Meshes for Interactive Modeling and Visualization	88
3 Subdivision Surfaces	91
3.1 Genesis of the Catmull/Clark Scheme	92
3.1.1 Evaluating points on a B-spline curve	93
3.1.2 Recursive Subdivision of B-spline Curves	94
3.1.3 Tensor Product Surfaces	97
3.1.4 Catmull/Clark Surface: Generalization to the Irregular Case	99
3.1.5 Rules for the Limit Position and Surface Tangents	100
3.1.6 Borders and Creases	102
3.1.7 Artist’s Delight: Summary of Vertex and Face Classifications	105
3.2 Practical Experiences with Subdivision Surfaces	106

3.2.1	Radius of Influence of a Vertex and a Face	106
3.2.2	Smoothing, Averaging, and Surface Shift	107
3.2.3	Linearity and Ripples	108
3.2.4	Non-convex Faces	109
3.2.5	Example Models	109
3.3	Options for Adaptive Tesselation and Display of Subdivision Surfaces	113
3.3.1	Recursive Subdivision with a Hierarchical Data Structure	113
3.3.2	Direct Evaluation: Parametric Surfaces and Basis Functions	115
3.4	Adaptive Tesselation on-the-fly of Catmull/Clark surfaces with Crease Edges	119
3.4.1	Vertex and Face Rings	119
3.4.2	Optimized Recursive Subdivision	121
3.4.3	Adaptive Realtime Display	127
3.4.4	Results and Discussion	128
3.4.5	Remarks on Tesselation Quality and Accuracy Issues concerning $k_{\max} = 4$	130
4	Practical Meshes	133
4.1	Triangle Meshes	134
4.1.1	“My First Triangle Mesh”: The Shared Vertex Data Structure	134
4.1.2	Shared Vertex Mesh with Traits	137
4.1.3	Halfedges and Mesh Iterators	140
4.1.4	Beyond Shared Vertices: Campagna’s Directed Edges and Hoppe’s Wedges	142
4.1.5	Automatic Mesh Simplification using Error Quadrics	144
4.1.6	Progressive Triangle Meshes	146
4.2	Boundary Representations and B-Rep Meshes	151
4.2.1	Skipvectors and Skipchunks	155
4.2.2	Polygon Triangulation	157
4.3	Combined B-Rep Meshes	160
4.3.1	Background on Combined B-Reps and Related Work	161
4.3.2	Combined B-Reps from a User’s Perspective: Manipulation, Update, and Rendering	164
4.3.3	The Combined B-Rep Data Structure	167
4.3.4	The Data Structure for the Tesselation of Smooth Faces	168
4.3.5	CommitUpdate, Tesselation, and Rendering	172
4.4	Progressive Combined B-Rep Meshes	179
4.4.1	Euler Operators for Mesh Manipulation	179
4.4.2	The Euler Operator Sequence	182
4.4.3	Euler Macros and Semantic LOD	183
4.4.4	Macro Culling	187
4.5	Mesh Modeling Tools	189
4.5.1	Converting a polygon into a double-sided face	189
4.5.2	Bridging two Faces	191
4.5.3	A Simple Extrude Tool	192
4.5.4	Path Extrusion	195
4.5.5	Intersection-free Extrusion and the Straight Skeleton	197
4.5.6	Refined Modeling Tools	209
5	The Generative Modeling Language GML	211
5.1	Putting the Pieces together	211
5.2	Language Introduction	216
5.2.1	The Language Rules	216
5.3	Examples for Shape Generation with Mesh Modeling Operators	221
5.3.1	Example Operator Chaining and Creating a Loop	221
5.3.2	Example Segment Intersection	223
5.3.3	Example Cube with a Hat using Raw Euler Operators	224
5.3.4	Example Arch or Door	226
5.4	Gothic Architecture	228
5.4.1	The construction of a Gothic Window	232
5.4.2	The Gothic Window in GML	237

5.4.3	Gothic Window: Results	240
5.4.4	The Procedural Cathedral	247
5.5	Discussion and Conclusions	255
5.5.1	The GML as a Generalization of the Known Ways of 3D Modeling	255
5.5.2	Persistent Naming and the Picking Problem	256
5.5.3	A new way to think about Shape? – Shape Understanding and Shape Complexity	257
5.5.4	Extending and Embedding the GML	260
5.6	Future Work – Fields of Application for GML-based Technology	262
5.6.1	3D Objects for Everybody	262
5.6.2	A Variety of Operator-based Shape Representations	262
5.6.3	GML + OpenSG to replace VRML/X3D: Scene Graph Scripting with Lazy Evaluation	263
5.6.4	Computer Games about Creativity rather than Destruction	263
5.6.5	A Double Layered Market for 3D Components	263
5.6.6	3D Support for all Computer Applications	264
5.6.7	True Generalized 3D Documents	264
5.6.8	The 3D Desktop	265
5.6.9	New forms of Human-Computer Interaction	265
5.6.10	Operator-based component technology	266
5.6.11	Integrating the GML into the Operating System Kernel	266
5.6.12	GML is stronger than XML	267
5.7	Epilogue	268
	Bibliography	270
	List of Tables	284
	List of Figures	285
	Index	291

Chapter 1

Introduction

1.1 The Shape Description Problem

None of the existing methods for describing the shape of three-dimensional objects is entirely satisfactory. These methods can be roughly divided in two classes. The first, much larger, class follows the ‘list of primitives’ approach. It describes three-dimensional objects and whole scenes composed of them as a – flat or hierachil – agglomeration of elementary geometric objects: Points, triangles, NURBS-Patches, spheres, cubes, blended blobs, and many others.

The second class is the class of procedural shape representations. It comprises a variety of different methods that may be less known, since none of them could gain general acceptance: Shape grammars and L-systems, functional composition, physically based simulations, shape programming languages, and several others. Also the proprietary scripting facilities provided by all modern software packages for procedural modeling and CAD fall into this class. The question why primitives still dominate today leads to a deep and important fundamental problem, the *shape description problem*:

“What is the right way to describe the shape of a three-dimensional object?”

Compared to the large body of literature on the different possibilities to represent the surface of 3D objects in a computer, only much fewer research was dedicated to this more general question. So far the only answer is ‘it depends’: When scanning, points are better, and CAD always uses NURBS. So the answer is technology driven, rather than following a thorough understanding of the ‘nature of shape’, especially of man-made shape. Many simple questions are still left unanswered that could give rise to more powerful shape representations as well as to more efficient modeling tools:

- How comes shape into existence? How is shape created, constructed, conceived?
- How does an artist or builder proceed? Which are the elements he or she combines in the first place?
- Which factors make that a shape can be recognized? Which are irrelevant and, thus, just artifacts?
- What is the ‘real shape’? What constitutes a shape class?
- Which objects are considered ‘the same’ by most people in all cultures?
- What is the ‘essence’ of a spoon, a chair, or a wind shield wiper?
- What is the shortest possible precise description of a shape? Of a whole shape class? And how are both related?

These questions are very general which makes them very hard to tackle. Sceptical fellow beings use to argue that solving these problems would also imply a solution of much harder problems, ranging from automatic shape matching to artificial intelligence, depending on the person’s respective background. This may be true; it is also true, though, that the fact that these problems were completely ignored for the longest time has brought 3D to the limits it currently has to struggle with. This will be illustrated in the next section; but it will also be shown that indications exist that maybe a completely new type of 3D technology is currently rising.

The objective of this thesis is not to solve the difficult problem of shape understanding. The goal is rather to provide a prerequisite for understanding shape, namely a – hopefully – general method to represent on a higher level of abstraction a shape that is already understood. This reflects the two tightly related aspects of the shape description problem: First, to find a methodology to formalize the description of shape and second, to assemble concrete shape descriptions for any given shape or shape class. The different aspects of shape description are now illustrated with a number of examples.

1000 chairs. This is the title of a very interesting book from Charlotte and Peter Fiell [FF97]. It contains indeed a collection of one thousand chairs from different times and different countries and, in particular, different styles. The variety of chairs is just amazing. It immediately leads to the fundamental question of how to characterize ‘a chair’.

The easy answer is the general answer: Anything a person can sit on. Indeed can a fallen tree or a stone in the right size be used as a chair. This is not a chair in the strict sense, however: A chair is usually manufactured for its purpose, it is movable, etc. But every attempt to give a more precise description of the shape of a chair seems to rule out all too many feasible chairs. After all there is a great difference between a comfortable armchair and a barstool.

An answer on an intermediate level on the scale between too general and too restrictive is an abstract one: The essential geometry of a chair is determined by exactly five points on its right half. They are mirrored to the left half to produce another five points, since chairs are symmetric. The five points are: two where the chair touches the ground, two that determine the right side of the seat, and one for the top right of the backrest (see Fig. 1.34 later on).

Architecture. There are millions of buildings in the world, and every single one of them is essentially unique. Even if two houses may be the same when they are built, they cease to be identical over time, as they are altered, remodeled, and converted. All buildings are nevertheless immediately recognizable as buildings. The reason is, of course, that they are all composed of more or less the same elements: Walls, doors, windows, and a roof.

A human observer can in most cases derive the purpose of a building alone from its shape. A church, a school, a university, a palace, a residential or an office building, they all have distinct shape properties. Architecture is a domain where also different levels of abstraction are used in parallel, and in the most natural ways. Architects use two-dimensional drawings for communication, usually orthogonal sections or views of the building (front, top, side view). On the highest level of abstraction, for city planning, individual houses are represented only as simple blocks.

But even when allowed to make only two dozen pencil strokes a good architect is able to convey the purpose of a building in a drawing. A very abstract view on architecture reveals the main methods to create a building from a number of simple box-like shapes: *attachment* and *containment* are the techniques at work when modeling with boxes.

Art nouveau and the issue of styles. What are the distinct constituents of a style? Art nouveau, also called *Jugendstil* or *Art 1900*, is a good example because it is a style that suddenly appeared, was very fashionable and trendy for some time, and gradually disappeared in the thirties. Art nouveau has penetrated all domains of shape, from architecture to clothing, sculpture as well as furniture, and the things of everyday life. So for each imaginable item there is an ‘art nouveau-version’: simple spoons and lamp shades, but also the various buildings from Antoni Gaudi in Barcelona, Spain – they all count as art nouveau. The most mystical fact is that all items that belong to this style are immediately recognizable as such. Art nouveau exhibits also another interesting feature, the development from art to craftsmanship. At a certain point, a skilled craftsman has ‘understood’ a style, and can faithfully reproduce it. A style is no item on its own; the distinct property of a style is that it can be applied to an item. It is a *transformation* that, given a shape, turns it into another.

System shapes. The domain of architecture brings in also another aspect, namely shapes that are composed of individual elements that fit together. Whereas these elements are created individually for most houses, the trend is to compose houses of customizable parts that can be manufactured more efficiently in an industrial assembly-line fashion.

One step beyond customizable shapes are *system shapes* made of a family of interoperable elements. They can be put together in various ways, very much like pieces of a construction kit, such as the famous *Lego* pieces [Leg]. *System furniture* for instance features shelf units that can be combined to line a wall in various ways; the same approach is pursued with success for cupboards, sofa elements, and kitchen furniture. Examples for system shapes from many other domains exist, just to mention the myriads of *system toys*, including also toy motor racings and toy railways.

The distinct property of customizable parts and system shapes, and the main reason for their great success, is that they radically *reduce the degrees of freedom*. Two *Lego* pieces can not be put together in arbitrary ways. They have to be axis aligned, and there is a regular grid of positions for the small hills, the *studs*, to snap together, as shown in Fig. 1.1.

The family of industrial shapes. Perhaps the most distinct example of a shape family with continuous degrees of freedom is the *screw*. Its primary parameters are the length and radius of the bolt, and the slope and the depth of the thread. The screw is also the basis for the most successful family of system shapes. Since the 19th century it is used for the assembly of all complex constructions in industrial engineering. Nearly all consumer products contain screws, since whenever attached parts are to be disassembled again there is no reasonable alternative to using them.

Today most industrial assemblies as well as consumer products are planned with the aid of computers. The basic operations are still the same: Parts are created by casting metal or plastic, they are treated by bending, milling, drilling holes, cutting, and welding, and the finished parts are finally assembled with screws. This is an example of a set of operations that is extremely well supported by high-end CAD and CAM tools because of its great practical and commercial importance. But it is by no means the only operator set one might reasonably want to create shapes with.

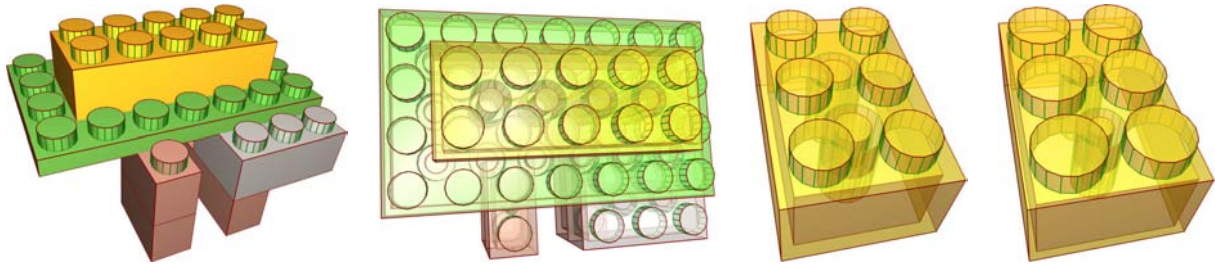


Figure 1.1: Lego pieces.

Variations: Continuous vs. discrete. Lego pieces can be snapped together only at discrete positions. Strictly speaking there is only a finite number of different models that can be built with a given set of Lego pieces. This is in harsh contrast to, e.g., the unlimited number of shapes that are possible with a piece of clay. Most interestingly, it is just this radical limitation of degrees of freedom that lets Lego unleash children’s creativity and motivates them to realize the shapes they imagine. The mechanisms of iterative shape design could certainly be studied very well systematically with the observation of children playing Lego: Right after finishing one shape they get the idea for the next modification by inspecting the shape they have just created.

Considering a Lego piece as a parametric object it is clear that its inner dimensions are not arbitrary: the pieces need to snap together reliably. The position of the studs is fixed, and their radius must be compatible with the width of the wall. Figs. 1.1 (c) and (d) show two different solutions to this problem. They suggest that there is a whole continuous 1-parameter-family of possible Lego pieces. The wall width is the free parameter here, which was varied, and the radii of the top and bottom studs are the dependent parameters.

Shape dependencies. This mechanism identifies one important, if not the principal, constituent of man-made shape: Dependencies. This is the *terminus technicus* for a whole number of meta-descriptions of shape: Proportion, symmetry, regularity, style, etc. – all these concepts have one principle in common, which is that ‘something’ is determined by ‘something else’.

The proportions of a greek temple determine how its width relates to its height and to the height of the columns as well as their distance from each other. Cars are symmetric, so car hull designers always design only one half of the hull. The bricks in a wall or on the pavement, the keys of a keyboard, the steps of a stairway, and innumerable other examples of man-made shape, they all exhibit striking regularity. They all contain sub-objects that are arranged not randomly, but according to some guiding rule. A style can be regarded as a set of a few high-level rules that permit to deduce many local decisions which are, thus, no longer free: the selection of colors, the absence or presence of decoration and ornaments, the choice of patterns and materials, all this is pre-determined by the choice of a style.

Technical constraints. Numerical and physically-based simulations, finite elements, statics, material properties, manufacturing methods, or simply cost – such hard constraints reduce the number of ‘free shape parameters’ even further, in addition to the aforementioned artistic principles. Both areas cannot be cleanly separated, though: One quality that makes a good artist or designer is his or her ability to transform a technical constraint into an artistic or design principle. The round arch is a central ingredient of the romanesque style – but it is also simply an extremely stable form of an arch.

According to the aforementioned measure, many ancient designers were extremely good, because so many ancient styles are made of transformed constraints. A notable example, which will also be further elaborated in section 5.4, is the Gothic style. It is an amazingly rich shape domain that is produced from only a limited shape vocabulary: All Gothic constructions are made of compass and ruler. The banal reason is that only then the plans could be reliably scaled from a small piece of paper to 1:1. The same constructions were simply executed with a larger compass and ruler.

The variation of rules as a general design principle. Two sources for shape dependencies have been identified so far: Aesthetic design and technical constraints. The notion of ‘dependency’ used here is a very general causal relation: It states that one shape ‘property’ is the consequence of another. In other contexts this may be called a *design rule*. It was also argued that defining rules is a central part of design in general. As a consequence a shape description method can only be adequate when it is capable of representing *shape rules* and *dependencies between shapes*. This is also the deeper reason for the fact that a list of primitives can never be an adequate method to describe shape.

The *rule variation conjecture* states that the same mechanism is at work also on the next higher level of abstraction: that not only objects vary as the result of applying a rule, but that also *rules vary* as the result of applying a ‘meta-rule’.



Figure 1.2: Structural similarity: The *linear sequence* as a fundamental mechanism of shape assembly.



Figure 1.3: Structural similarity. Buildings in the *Wilmerdingstrasse* in Braunschweig, erected 1890-1900.



Figure 1.4: Structural similarity. The *radially symmetric sequence* as a fundamental mechanism of shape assembly.



Figure 1.5: Structural similarity. Despite their diversity all Wilmerding buildings share some essential properties.

The sequence as an abstract shape generation pattern. The images on the previous two pages shall illustrate the hypothesis that *shape design* is to a large extent *rule design*. Rules are the basis for creating patterns.

The first step to decipher a rule is to discover an abstract pattern that a collection of everyday shapes has in common, for instance the *linear sequence*. Most of the examples in Fig. 1.2 are actually fences. Some of them contain a horizontal sequence of vertical elements; others are two such sequences intertwined or nested, and sometimes a systematic variation is applied to the individual elements. Others yet are made of a massive piece, and distributed are holes rather than elements. And finally, when it is possible to create a horizontal sequence of items, it should as well be possible to create a vertical sequence of horizontal sequences – which results in a regular grid of items.

The second step is the discovery that a linear sequence can be applied as well radially as, e.g., the collection of hubcaps in Fig. 1.4 suggests. A hubcap can be understood as a completely regular linear sequence of items, only transformed to fit on a circle. It is even more regular than the fences because of the necessity that the center of gravity is exactly in the center of the circle. Note here again the nice contrast between distributing items or holes.

Building facades as multiple nested sequences. The most complicated examples, and a rich source for a whole number of different design principles, are the images of façades in Figs. 1.3 and 1.5. It is most interesting to examine how each of the six displayed houses develops its own distinct style. All of them are different, but they also exhibit striking similarity; for instance they are all symmetric, they are four stories high, and they all vary their styles from floor to floor. Not the ground floor but the first floor has the richest decoration. The horizontal symmetry is twofold: The right and left halves of the facade are symmetric, but the window columns are also grouped in pairs. Note that most houses have eight columns, and note the different forms of pairwise grouping. – The overall structure is determined by all sorts of nested sequences.

Textual shape descriptions. To describe shape textually is a frightening task. The reader might imagine a person right in front of a highly complicated shape, such as a crumpled up handkerchief, or a Gothic cathedral. The man is equipped only with a type-writing machine, with which he is urged to produce an *unambiguous description* of the shape he is facing. The description may contain only text, in any human or formal language, but no images or drawings. It has to be, however, so exact that it would permit to faithfully reproduce the shape under consideration.

This scenario describes another version of the shape description problem. Computers are machines for symbol manipulation, a computer program transforms input to output. Concerning 3D shapes, the output might be a rendered image, which is nothing but a regular grid of samples of the surface. – But what is the input?

The task of designing a 3D file format is to define a suitable formal language for expressing shape descriptions. The scenario makes the requirements of such a language immediately clear: The person perceiving a shape, understanding a shape, now wants to express his or her idea of the shape. Even if the formal language is hidden behind a graphical user interface, it has to capture the whole description when it is used as a file format. So what counts is the language.

There is, of course, a big difference between a crumpled up handkerchief and a Gothic church: The handkerchief is usually not consciously formed. For the construction of the church, though, it was necessary that many individuals share the same very precise idea of a not yet existing shape. – How did they receive it? And how was the plan executed?

Other forms of shape communication. One might argue that the simplest way to derive a formal language is to systematically review existing examples of textual shape descriptions. Yet remarkably few such descriptions exist that are precise enough. Shape is most often described only by alluding to prior knowledge. When talking about a specific chair it is assumed that general knowledge about chairs is readily available and can be referred to, so that only the specific new properties need to be formulated explicitly. The computer does not have this implicit knowledge, though.

One might argue that text is inappropriate for describing shape. The primary communication channel for shape is via drawings, sections, plans and images. But they all share the same problem with text, although this fact is more effectively hidden away: To read a 2D plan requires implicit prior knowledge about shape. A 2D plan is not an unambiguous description of a 3D shape, 2D plans are merely coded suggestions that appeal to human imagination. A computer can not process the front, side, top plans of any complex 3D object automatically and reconstruct it. And even if, this does not solve the essential problem of any artifact description method that only alludes to the ‘real thing’: What are the ideas behind the plan? What are the design rules that the plan is derived and instantiated from? Which plans are ‘similar’?

Procedural models deserve a procedural representation. This claim summarizes what has been discussed so far. It was attempted to underpin the eminent importance of rules in the process of shape creation. Rules are present in every shape around of us. Rules and dependencies correspond to the natural way human beings communicate about shape: This is just as long as this one, that is attached, and this is the same as that except for this and that.

It is evident the mentioned facts are highly relevant for the definition of suitable digital file formats for three-dimensional shapes. The discussion so far remained on a rather abstract level; in the next section it shall be concretized. It shows that the theoretical shortcomings of the ‘list of primitives’ approach indeed result in very concrete practical problems.

1.2 Review of Applied Shape Design and Interactive Computer Graphics

The previous section has discussed different aspects of the shape description problem on a more theoretical level. Of course it has also very practical aspects since whenever shape is generated, designed, acquired, transmitted, or used, it always has to be *described*. One very important aspect was not mentioned so far, though: When using a digital computer to design shape one would also like to see this shape. So there is a tight relation between shape design and interactive rendering. With current technology this is all too often a one-way-road: first the design, then the visualization. This is the source of many problems, as will be shown in the following.

This section has two objectives. Its primary purpose is to summarize roughly the current state of 3D technology, and especially to highlight its relation to the problem of shape description. These considerations shall motivate the new solution proposed in this thesis, which is for the time being referred to as the *generative* or the *GML approach*. It will be de-mystified in section 1.6 and, of course, explained in detail in the remaining chapters.

Interactive visualization is a new quality of 3D. First of all there are two fundamentally different ways to present computer-generated imagery, namely as *pre-rendered films* or as *interactive visualization*. Offline rendering can make use of all possible rendering features. It may employ time-consuming techniques such as raytracing and radiosity to faithfully approximate the global illumination in the scene in every frame. The computation of a single image of a movie production typically takes hours to complete, which is why whole *render farms* are employed for this purpose.

Interactive visualization, on the other hand, requires that at least 20 images are rendered *per second*. This means that the computer has got only 50 milliseconds, or less, to generate the whole image. With 20 fps, *frames per second*, a feeling of fluent interactivity emerges. With lower frame rates the latency between user interaction and visual feedback gradually leads to a perceived loss of control and precision of motion.

The difference between offline and online rendering is not only a quantitative one. Theorists use to argue that what really matters for the quality of algorithms and data structures are complexity classes; there is no such thing as Moore's law for Turing machines. But at some point also a quantitative increase may suddenly turn into a qualitative change, simply because something becomes possible that was not possible before. For computer graphics, this situation arose around the year 2000, when high-end 3D hardware became affordable all of a sudden. At the same time the CPU speed surpassed the 1 GHz barrier, and substantial computing resources became available per frame. Today each and every newly sold computer is equipped with powerful 3D hardware, even office computers and cell phones. This remarkable development was stimulated by the games industry, which has since then gained great commercial significance.

The 3D software crisis. The usefulness of 3D goes far beyond computer games. But so far no other, more serious, *killer application* has unleashed the undeniable potential of 3D on a mass-market scale. Using 3D has of course become much simpler for larger and smaller companies. It is employed whenever possible in product design, advertising, special effects etc. But all these are primarily offline techniques, where 3D technology is used as a device for creating stunning movies.

The fundamentally new quality of computer-generated 3D, compared to a movie, is that the user can influence the non-existing, *virtual*, world that is presented to him. The communication is no longer one-way, the consumer can now feed back to the multimedia experience. Interactive 3D was deemed to change the way people interact with computers; but Web3D (3D on the internet) and VRML as its most prominent technology, have failed to do so. What was the reason?

One problem might be that it is not fully understood what it means to *interact* with a 3D scene. The new quality of *interaction* might also require qualitatively new approaches, since the old approaches no longer scale: Low-level interaction is tedious. More powerful interaction is more efficient and, thus, reduces the cost. And only when 3D becomes cheaper and more powerful it will be applied also by non-expert users in more application areas than today.

Separation of modeling from viewing. The situation for visualization and interactive rendering today is characterized by a fundamental dichotomy, the separation of the model creation from the interactive visualization. Unfortunately the same type of technology is employed for creating movies as well as for interactive 3D. In either case the proceeding is basically the following:

1. Modeling – either by shape acquisition (real shapes) or by manual modeling (synthetic shapes)
2. Export – to some 3D interchange file format
3. Rendering – to either to create a movie (offline rendering) or by interactive techniques (online rendering)

A computer can generate images only when it has something to display. The 3D objects that make up the scene need to be somehow transferred to a digital form. They can either be existing real objects, or they are imaginary synthetic objects. Correspondingly there are basically two methods to obtain digital shapes: shape acquisition and shape modeling.

1.2.1 Shape Acquisition

The term *3D scanning* suggests that shape acquisition is a standard method that can be applied ‘out of the box’; that it works reliably, fast, and automatically, very much like a fax machine. Indeed a prominent scanning initiative initiated by Mark Levoy at Stanford, USA, enthusiastically started with the idea of a ‘3D fax machine’ [3DF94]. As it turned out soon, the truth is that 3D scanning is a very complicated craft. To obtain good result requires substantial efforts which is also witnessed by, e.g., the vast experiences of the group from Roberto Scopigno in Pisa, Italy [BCF*04, CCG*04]. The following are the practically most relevant methods, ordered according to hardware cost:

- **Photogrammetry** is the reconstruction of shape from multiple photographs. This involves to reliably identify corresponding *feature points* in the images. The disparation permits to infer the relative depth of each feature point. This technique is inexpensive and it works with digital photos as well as with video sequences. Disadvantages are the low precision and the density of the resulting point cloud are not very high, and that not all objects exhibit distinguishable feature points [PGZF01, LCZ99].
- **Structured light** is typically employed under controlled lighting conditions for acquiring smaller-scale objects that can be put, e.g., on a turntable. Specific patterns are projected on the object using, e.g., a digital video projector, and then a series of photographs is taken. Since the pattern is known, as well as the orientations of projector and camera, a dense grid of points on the surface can be accurately reconstructed, roughly up to one million per scan, with sub-millimeter accuracy.
- **Laser-range scanning** proceeds by radiating a grid of laser rays from a sophisticated device that measures, for each ray, the time until the reflected light returns. A range of different devices exist which, in a single scan, can measure up to several million points with high accuracy. The resulting regular grid of points provides also implicit connectivity. Problems are very reflective and translucent materials that reflect no light at all, or different rays at once, to the device, as well as noisy and scattered surfaces such as hair, tree leaves, and also windows.

More specialized approaches for shape acquisition include computer tomography (MRT,CT) for capturing volumetric data, as routinely used in medicine, and taking explicit surface samples, e.g., with a haptic feedback devices such as the [phantom](#) device and other measuring systems. They are especially important in industry for checking finished parts. – Most objects can not be acquired with a single scan. Multiple scans from the same object need to be further processed by

- *scan registration*, i.e., transformation into one coordinate frame in a compatible way, and
- integration into a common mesh, which involves removal of outliers and a *re-meshing* step.

When the scan was inappropriately planned it may then be that parts of the surface are missing. Furthermore all optics-based acquisition methods suffer from the fact that only visible portions of the surfaces can be scanned; cavities and occluded parts always lead to holes in the mesh.

What is the point? The implications and inherent limitations of 3D scanning become obvious when comparing it to its 2D analogon, digital photography. A scan is a frozen grid of samples from the surface of a three-dimensional object. It can be understood as some form of *multi-view photograph*, since computer graphics permits to synthesize an unlimited number of new views from a scanned 3D object. This property makes 3D scanning ideal for documentation and archival purposes, very much like photos, since it is a decent recording of the state of an object’s surface at a certain time.

Scanning is a ‘blind’ process. The the raw scan points are arranged in a regular grid, very much like the pixels in a photo. It makes no difference whether scanning a piece of jewelry or a table top, even though the geometry of the latter could be perfectly represented by one quadrangle instead of a million triangles. This is a drawback but also a feature of the method: All the small local imperfections in the table are faithfully represented in the data. This leads to a practical problem, namely the huge size of the scanned datasets, which is even increasing as the precision of the hardware advances. Methods were developed for the reduction of mesh sizes [IG03]. The size of a scanned 3D dataset, however, depends on the question how the data are represented and is, thus, directly related to the shape description problem.

The use of a scan, and shape matching. What is a scan good for beyond archival? The fact that a scanned object can be interactively inspected very strongly suggests to most users that its shape is now captured by the computer. This is not so: A scanned desk lamp has lost its ability to move, a scanned screw has become one with the object it attaches. Scanned toy cars have their wheels fixed forever; even the cup on the table becomes one with the table when scanned.

Most objects are not flexible or have hidden mechanics. But the *segmentation problem* is serious: A single triangle does not know whether it belongs to a door, to a window, etc. It may sound like a strong statement but it is probably true that that *to recover semantic information from a scan is a prerequisite for any further non-trivial shape manipulation*. Of course there is a whole range of possibilities to retrieve semantic information, in accordance with all the different levels of shape description, from local shape features to the function a 3D object serves. The re-usability of scanned data is a

problem. It is very difficult to retrieve, e.g., only the chairs from a large set of scanned interiors. This involves to solve the *shape matching problem*, which can be stated simply as: **Given a shape, find all similar shapes**. Attempting to solve the problem in this very general form is of course somewhat naïve. Accordingly the results are still rather limited [NK03]. This is another indication for the depth of the shape description problem. Shape matching involves to answer the question: What is a chair?

1.2.2 Shape Modeling with Procedural Modelers

The main device for creating synthetic shapes on a computer is 3D modeling software, also called a *modeler*. High-end modeling software is usually extremely complex, provides many pages full of features, and comes with numerous manuals and tutorials that demonstrate how to create *virtual*, i.e., non-existing, sceneries. Modeling software can be roughly divided in two classes: Procedural modeling tools are more targeted at movie creation, while CAD modelers are for high-precision industrial shape design.

There is a bit of confusion in the nomenclature: The term *modeler* denotes in some cases the modeling software, and in other cases it means the person operating this software, i.e., the user of a modeler. In order to avoid this confusion the operator is called *artist* or *designer* in this thesis, which appear to be the terms more and more used in the literature.

Fully fledged allround 3D modeling and animation packages are targeted at creative 3D shape artists for creating motion pictures. The typical application areas are (i) fully digital animated movies and cartoons, and (ii) short special effects sequences (*SFX department*) to be overlaid with filmed real scenes (*compositing*). While originally employed as a substitute for film takes that are difficult to realize, SFX have become ubiquitous today in commercials and advertising. Shape design, which is the focus of this thesis, is only one among several aspects of the digital film workflow. 3D Software is typically arranged more like a virtual film set or a 3D film studio, which involves the following modules:

- **Shape modeling** is the process of creating the shapes of all the objects in a digital scenery. Typically this comprises buildings as well as furniture, vehicles, and vegetation, but also physical phenomena such as water, rain, fire, etc. Surface models of animals and characters are attached to skeletons, sets of bones, to control the shape deformations. The models are created using procedural tools, consequently all models are *procedural models*.
- **Material assignments** determine the look of the virtual world. Materials with complex reflectance properties can be defined with *shader networks*. Materials may comprise also textures, which requires texture coordinates on the surface. *Detail textures* can be applied to add small-scale high-frequency surface details to enhance the shape.
- **Animation** is the definition of motions in the scene. All of the object and material parameters can usually be animated, i.e., they may vary as a function of time. Each scalar parameter is typically animated using an editable spline curve, i.e., three of them for the three *channels* (x, y, z) of an animated position (or rotation).
- **Rendering** finally is the process of image generation. It can be very time-consuming, but it works usually fully automatically in *batch mode*. High-end productions employ whole *rendering farms* for computing, e.g., the global illumination in a scene, which is basically an $O(n^2)$ problem for n surface patches (or triangles).

Usability is key for artists who are operating the same software for hours and weeks: Swift interaction, navigation, selection modes, access to tools, configurability of menus and keyboard/mouse behaviour etc. are vital. The general approach to shape modeling of the different packages are very similar, only the (long) feature lists vary from software to software. The artist needs to get acquainted with all the features of the chosen tool. To some extent, this experience can also outweigh the deficiencies of the software: Rather than switching over to a new, optimal, software all the time, artists apparently prefer to stick to their favorite modeling software that they know so well.

The forward modeling style. The name ‘procedural modelers’ comes from the approach to shape modeling these tools have in common. Artists successively apply sophisticated modeling tools to alter a shape. The shape is changed until it matches the the ideas of the artist and the specifications. Whenever the specifications change, however, the shape is *further* deformed and manipulated until it matches again. This is fundamentally different from a parametric modeling style where only the appropriate modeling feature is updated to accomodate the desired changes. More complicated shapes therefore require decent planning, which parts are to be created first, then refined, etc. Modeling and sculpting are crafts that simply require much practical experience.

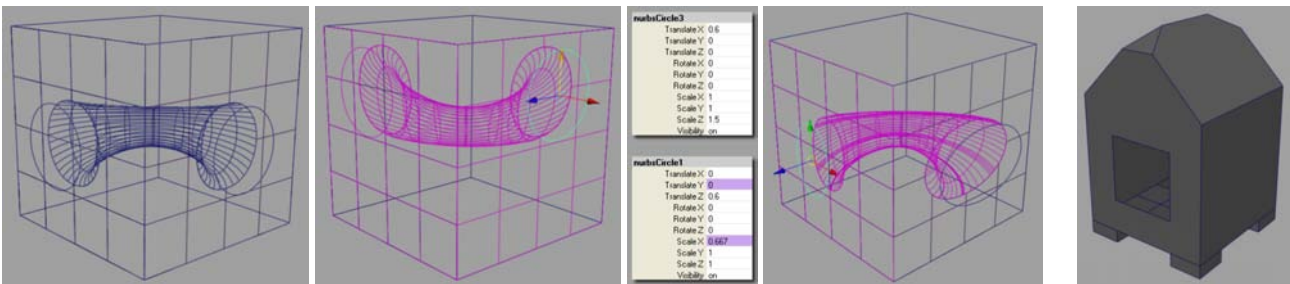
To some extent the market is dominated today by the two high-end packages *3D Studio Max* from Kinetix and *Maya* from Alias. Some of the modeling tools with wider spread use are listed in the table in Fig. 1.7. They can be compared through downloading trial versions. It is worthwhile to have a decent look at their feature lists: Most artists admit that in practice they use only a small fraction of their favorite tool’s modeling functionality. –

The author’s personal recommendation for a 3D sketching tool for casual users is not in the table. It is called *Sketchup*, available from www.sketchup.com, and it is rather ‘lean and mean’. Sketchup has few but well-chosen modeling operations, an interesting approach for interactive handling, is easy to learn and use, and it yields very clean meshes.

```

1 nurbsCube -p 0 0 0 -ax 0 1 0 -w 1 -lr 1 -hr 1 -d 3 -u 1 -v 1 -ch 1 ;
2 circle -c 0 0 0 -nr 0 0 1 -sw 360 -r 0.2 -d 3 -ut 0 -tol 0.01 -s 8 -ch 1 ;
3 move -r 0 0 0.6 ;
4 circle -c 0 0 0 -nr -1 0 1 -sw 360 -r 0.1 -d 3 -ut 0 -tol 0.01 -s 8 -ch 1 ;
5 circle -c 0 0 0 -nr 1 0 0 -sw 360 -r 0.2 -d 3 -ut 0 -tol 0.01 -s 8 -ch 1 ;
6 move -r 0.6 0 0 ;
7 projectCurve -ch true -rn false -un true -tol 0.01 "nurbsCircle1" "leftnurbsCube1" ;
8 projectCurve -ch true -rn false -un true -tol 0.01 "nurbsCircle3" "frontnurbsCube1" ;
9 loft -ch 1 -u 1 -c 0 -ar 1 -d 3 -ss 1 -rn 0 -po 0 -rsn true "leftnurbsCube1→projectionCurve1_1" \
10 "nurbsCircle2" "frontnurbsCube1→projectionCurve2_1" ;
11 trim -ch on -o on -rpo off -lu 0.1 -lv 0.1 leftnurbsCubeShape1 projectionCurve1_Shape1 ;
12 trim -ch on -o on -rpo off -lu 0.1 -lv 0.1 frontnurbsCubeShape1 projectionCurve2_Shape1 ;
13 expression -s "nurbsCircle1.translateY_=_nurbsCircle3.translateY" -o nurbsCircle1 -ae 1 -uc all ;
14 expression -s "nurbsCircle1.scaleX_=_1/nurbsCircle3.scaleZ" -o nurbsCircle1 -ae 1 -uc all ;

```



```

1 polyCube -w 1 -h 1 -d 1 -sx 1 -sy 1 -sz 1 -ax 0 1 0 -tx 1 -ch 1 ; // create poly cube
2 polySplit -ch on -s 1 -sma 0 -ep 7 0.5 -ep 6 0.5 pCubeShape1 ; // split face make edge
3 select -r pCube1.e[14] ; // select and
4 move -r -y 0.5cm ; // move edge
5 polyCreateFacet -ch on -tx 1 -s 1 -p -0.5 0.25 0.25 -p -0.5 0.25 -0.25 \
6 -p -0.5 -0.25 -0.25 -p -0.5 -0.25 0.25 ; // create plane
7 duplicate -rr ; // duplicate and
8 move -r 1 0 0 ; // move
9 polyUnite -ch 1 pCube1 polySurface1 polySurface2 ; // combine faces
10 polyMergeFacet -ch on -ff 5 -sf 7 polySurfaceShape3 ; // make hole
11 polyMergeFacet -ch on -ff 4 -sf 7 polySurfaceShape3 ; // make hole
12 select -r polySurface3.vtx[8:9] ; // select and
13 polyChamferVtx 1 0.25 0 ; // chamfer vertices
14 polySoftEdge -a 0 -ch 1 polySurface3 polySurface3.e[*] ; // make all edges hard
15 polySubdivideFacet -dv 2 -m 0 -ch 1 polySurface3.f[3] ;
16 select -r polySurface3.f[18] polySurface3.f[21] polySurface3.f[12] polySurface3.f[15] ;
17 polyExtrudeFacet -ch 1 -kft 0 -pvx 0 -pvy -0.5 -pvz 0 -tx 0 -ty 0 -tz 0 -rx 0 -ry 0 -rz 0 \
18 -sx 1 -sy 1 -sz 1 -ran 0 -divisions 1 -twist 0 -taper 1 -off 0 -ltz -ws 0 \
19 -ltx 0 -lty 0 -lrx 0 -lry 0 -lrz 0 -lsx 1 -lsy 1 -lsz 1 -ldx 1 -ldy 0 -ldz 0 \
20 -w 0 -gx 0 -gy -1 -gz 0 -att 0 -mx 0 -my 0 -mz 0 -sma 30 \
21 polySurface3.f[12] polySurface3.f[15] polySurface3.f[18] polySurface3.f[21] ;
22 setattr "polyExtrudeFace1.localTranslate" -type double3 0 0 0.1 ;

```

Figure 1.6: Examples of the *Maya Embedded Language* (MEL) from Alias/Wavefront. The code was generated by Maya in background during an interactive modeling session of Matthias Richter. The code generation feature is supposed to facilitate the authoring of shape families and re-usable shapes. Parameterized shapes can also be manipulated with interactive gizmos, but only within a running Maya. – Note the compactness of a procedural shape description versus the unlimited number of shapes it can produce.

The first example (images a-d) creates a NURBS cube into which a NURBS pipe is inserted. The pipe connects two circles that are trimmed out of the cube sides. The height and radius of the circles can be controlled interactively with an arrow gizmo using expressions. Note the (implicit) numbering of the created entities, which is one of the obstacles encountered when trying to write re-usable MEL scripts.

The second example demonstrates mesh modeling on the half-edge level in a MEL script. It creates the little house in image (e) from a cube whose top face is split by inserting an edge which is subsequently moved (lines 1-4). The opening is inserted by trimming out a quad (lines 5-11). Extrusion is a very important modeling tool because of its versatility. This is documented by the great number of options for it in line 17 where the four feet are extruded. Note the use of explicit face indices 12, 15, 18, and 21 in the generated MEL code, and compare this to the considerations in section 1.2.3 concerning the *persistent naming problem*.

Maya	www.alias.com	AutoCAD	www.autodesk.com
3D Studio Max	www.discreet.com	Dassault Catia	www.3ds.com
Houdini	www.sidefx.com	Pro/Engineer	www.ptc.com
Rhinoceros	www.rhino3d.com	SolidWorks	www.solidworks.com
Cinema4D	www.maxon-computer.com	UGS SolidEdge and NX	www.ugs.com
Lightwave	www.newtek.com	MicroStation	www.bentley.com
Caligari truespace	www.caligari.com		
SoftImage XSI	www.softimage.com		

Figure 1.7: Typical procedural modeling tools (left) and high-end CAD systems (right) in use today.

1.2.3 Shape Modeling with Parametric CAD Software

Software for *computer-aided design* (CAD) differs from procedural modeling tools in that it provides very high, *guaranteed* precision, and it integrates seamlessly with the industrial workflow in the context of computer-aided manufacturing (CAM). It is very important to assert that the modeled objects are indeed manufacturable. Another distinct high-end feature is *product data management* (PDM). It is concerned with versioning, product life cycle management (PLM), metadata administration, and all the other services necessary for maintaining manufacturable industrial products composed of many different parts: Data need to be brought consistently from the design to the CNC-milling-machine, individual parts from different sources must be assembled for digital mockups (DMU), and PDM/PLM also has to assert that the part database is consistent with the printed documentation.

The high-end CAD market is firmly dominated by the ‘*major CAD companies*’ listed in Fig. 1.7 to the right. They provide high-end functionality at a high-end price; but they usually offer good conditions for academic and educational institutions. The very remarkable thing about these software packages is that probably the vast majority of all consumer products in the world is designed using one of them, as was pointed out by Ari Rappoport during SMI 2004 [SR04].

Parametric design is still new to some. Since the late 1980’s and early 1990’s a veritable revolution took place in high-end CAD, namely the introduction of *parametric design*. This novel and innovative approach originated from the *Pro/Engineer* system from *Parametric Technologies* (listed in Fig. 1.7, also see the wikipedia article [wikb]). Due to its enormous success the method was quickly adopted by the other major CAD vendors. Today all high-end CAD packages support parametrics in one form or the other. – Rappoport further explained, and complained, that the advent of parametric design has been largely ignored by the academic community. No systematic research takes place, no decent comparison of approaches, and it appears that the possibilities of this new field of design are not explored, except, since it is a multi-billion dollar market, by commercial companies. Most computer graphics professionals from academia admit that they have no or very limited practical experience with high-end modeling/rendering software packages. Developing new algorithms is much more en vogue than to reflect on the existing approaches. Shape modeling is sometimes disregarded as being a craft rather than a subject to science. Furthermore it is extremely tedious to compare feature lists of modeling packages. There is the risk of doing only software studies, and generalizability is an issue there. Of course there are many interesting detail problems. But do they indicate shortcomings of the general approach or just of the particular software?

Another reason for the reluctance of academia might be the variety of different proprietary approaches. Although the basic idea is the same, each tool apparently realizes parametric design in a slightly different fashion. There are also many different words for it. Sometimes it is labeled ‘variant’ or ‘variational’ design, other tools call it ‘mechanical CAD’ or ‘associative design’. In any case, the idea is that every part created, every shape feature, and every modeling operation knows about its dimensions and its relation to other parts. The *degrees of freedom* (DOF) can be iteratively fixed by defining relations, *constraints*, until the whole assembly is fixed. This process is supposed to resemble the way machines are constructed in practice: When attaching a part to another with a screw, only a single rotational DOF remains.

Associative design is similar to the process of *dimensioning* where little labels are set into a 2D drawing to say how thick the wall is, how far the centers of two holes are apart, etc. Dimensioning is a creative process since, in order to keep the drawing readable, only the essential dimensions are conveyed in a plan. So part of the design process is to find out which dimensions are the essential ones. This information can be exploited by doing the reverse, i.e., to derive the distance of the holes from the dimensions specified by the user.

Intelligent 3D components are another form of parametric design that can be primarily found in software for rapid architectural prototyping (*house planning software*). It usually comes with a component database providing, e.g., a staircase that automatically adapts when the distance between floors, the *floor height*, is changed. The same applies to windows and doors that adapt to wall thickness, respect inner walls, etc. The great problem is only that the parts database is pre-fabricated so that it is not possible to add user-defined intelligent 3D components to the parts database.

Background on parametric modeling. Parametric design has not been completely ignored by the academic community, of course; a bit of material exists. A good entry point is the *Handbook of CAGD*, especially the chapters 20, *Solid Modeling* from V. Shapiro and 21, *Parametric Modeling* from C. M. Hoffmann and R. Joan-Arinyo [Sha02, HJA02]. They list about one hundred papers on parametrics, many of them by researchers who are very close to the production industry.

Shapiro points out that “parametric families of solids is a widely accepted, but poorly understood, notion”; Hoffmann and Joan-Arinyo assist in saying that “there is no satisfactory definition of the term *variational class of solids*”. These statements support the hypothesis from the introductory section 1.1, which discusses the shape description problem in the wider context of man-made shape: A formal definition of a shape class can only be satisfactory when it captures the human concept of shape design as rule design.

The main components of industrial parametrics are *features*, representing geometry, and *constraints*, representing relations between feature parameters.

Constraint-based modeling. The vision is that a designer can make a quick sketch of a machine part using standard geometric primitives like lines, circles, etc. In a second step, the sketch is converted to a technical drawing as the designer specifies how the primitives relate to each other, e.g., that a circular arch is connected to a line segment so that both have the same tangent. Accordingly, there is a large number of possible constraints. They can be roughly partitioned into four classes: (a) *geometric constraints* such as parallelism, tangency, symmetry, coaxiality, concentricity, perpendicularity, distance, angle, (b) *equational constraints* such as, e.g., torque, (c) *semantic constraints* that assert the validity of a construction, and (d) topological constraints: attachment, containment, incidence, connectivity. The constraints define a *constraint graph* where edges relate parameters of geometric primitives. The remaining free parameters are supposed to be determined by employing automatic *constraint analyzers*. Some dimensions are the result of others (‘length overall’), or can be derived by some reasoning (automatic *constraint solvers*).

One problem of constraint-based modeling is that realizable drawings may not be over- or under-constrained. Constraint solving involves inference and numerics which makes it not very transparent. Users report that small parameter changes sometimes lead to unexpected, discontinuous results in parametric modeling systems.

In the context of the introductory section 1.1 another, perhaps more fundamental, objection to constraint-based modeling arises. Constraints can only relate feature parameters that exist in the parametrization offered by the modeling system. But the great difficulty, and the key to a re-usable shape description, is just to determine the ‘right’ parametrization of a given shape *in every situation*. There is an unlimited number of ways to parameterize a simple rectangle, or a box. As pointed out by Pratt in [Pra04], reparametrization is key: The STEP standard defines a circle in 3D using (a) midpoint, axis, and radius. But some parametric modelers offer also parametrizations through (b) three points, (c) tangency to three other circles, (d) tangent to two other circles plus radius, etc. This problem can only be solved when the system allows the designer to define new parametrizations.

Feature-based modeling. This is the alternative to constraint-based modeling. Feature-based design is an enhancement of its predecessor, *variant design*, where objects are described with symbolic statements like BLOCK(w,h,d). The 3D object is created as a *side effect* of evaluating the statement. The advantage is that parameters can be replaced by arbitrarily complex *expressions*, and the modeling statements can be embedded in some kind of scripting language environment. Apparently this approach was first pursued in 1982 by Brown with his PADL-2 system [Bro82].

The term *feature* is used in various contexts, so it is defined only rather sloppily as *a generic shape with which engineers associate certain properties or attributes and knowledge useful in reasoning about a product* [HJA02], 21.5.1. The constituents of a feature are (a) generic shape (B-rep etc.), (b) behaviour (attributes like parameter ranges etc.), (c) engineering significance (information to downstream applications etc.). Feature models can be constructed (a) a posteriori by grouping and annotating objects in an existing geometric model, (b) by automatic feature recognition, or, (c) as a design methodology, called *design by features*, which is the predominant method for shape design in a feature-based parametric modeler. Ideally the features are shape templates from a library of application-oriented, user-defined features.

Another issue concerns the question of how to represent shape features. Two general approaches exist; the *declarative representation* uses again geometry plus constraints, whereas the *procedural representation* describes explicitly how to build a feature, instead of the implicit description specifying a set of conditions the feature has to fulfill. But as Hoffmann/Joan-Arinyo state (with respect to variant design), “design as programming is less desirable than giving the designer visual tools and deriving, from a visual design process, a flexible and intuitive parametric design” [HJA02], 21.4. This *code generation problem* is of great practical importance, and it will re-appear several times in this thesis.

The problem of generating different views on the same model. The same model may be used in different contexts, which also require *different geometry*; the design view, for instance, is different from the manufacturing view. A simple example is a through-hole in a part which is then divided in two holes by inserting a supporting feature into the middle of the hole. In reality, though, the supporting feature is not added, but is simply not removed when the hole is created by a

CNC milling machine ([HJA02], Figs. 21.2, 21.3). More complicated examples occur, e.g., in the automotive industry. For a car there is (a) the design manipulation view, exterior and interior, (b) the design review model, optimized for interactive visualization, (c) the model for crash test simulations, (d) the model for the virtual wind tunnel, plus several others. All these models of the same object differ not only in the level of geometric detail, but they also put different emphasis on the various parts that make up the car. The difference is of a semantic rather than just a geometric nature.

The problem of changing views is again directly related to the shape description problem: When talking about different views of the *same* car, this raises immediately the question what ‘the car’ actually is. It would be ideal, of course, if there was one ‘master model’ from which the other views can be derived, so that changes are immediately propagated and conflicts can be resolved in the master model. But how can these derivation rules be described?

The persistent naming problem and the picking problem. The problem of persistent naming can be stated simply as follows: *What is a part?* – Whenever applying a modeling operation, a feature, or a constraint to a model, it is necessary to unambiguously refer to the location on the model *where* it is to be applied. Typically this is done by, e.g., interactively selecting a vertex, an edge, or a face using a 2D pointing device, a mouse. The advantage of parametric design is that model parameters can still be edited, and the model is simply re-generated accordingly. Now the question is: Where is the selected vertex, edge, or face? It may be that one, or several, or even no entity at all correspond to the original one when the model is re-generated in response to the parameter change.

According to Shapiro, the persistent naming problem reduces mathematically to the difficult problem of indexing the connected components of an implicitly represented set [Sha02], 20.7.1. According to Hoffmann/Joan-Arinyo, the persistent naming problem is not solvable in a general way [CCH96]. They propose a more modest approach, namely to postulate that any ‘successful semantics’ for parametric shape design must fulfill two properties:

- Model instantiation must be *continuous*, i.e., small edits should yield small changes, and
- it must be *persistent*, i.e., after returning to the original parameter values the original design should reappear.

In the context of the shape description problem, the problem of persistent naming is very much also a *picking problem*. It is most likely that a designer has reasons for picking exactly a particular entity. This means that there is some underlying rule. If the designer only picks an edge but does not tell the rule, the software can only speculate about it, which is just the reason for the insufficient stability of the selection after re-evaluation: A particular step of a stairway might have been selected because it is the 14th step, or because it was at height 2.5 meter, or because the staircase approaches a wall, etc. The selection rule itself, and not only a particular selection, needs to be included in the shape description.

Intricacies of CAD model exchange. A serious practical problem is caused by incompatible proprietary parametrics. As it is explained by Michael J. Pratt, who provides valuable background on the problem in his SMI04 paper [Pra04], the typical CAD exchange standards, STEP and IGES [ste94, IGE96], were defined before the advent of parametrics. When creating an intelligent part X in Catia it loses its intelligence when loaded into SolidWorks; the same is true for all other combinations. Only a ‘frozen’ instance x of X can be exported to SolidWorks. This is disastrous, of course, when creating a *digital mockup*, a complete digital assembly of a whole new machine composed of many parametric parts. In case the parts come from different companies and are created with different tools, the complete assembly can not use parametrics any more for adjustments. Parameter changes can only be done in the source systems, and the changed instances need to be exported again to STEP or IGES as a ‘dumb’ model, without any parametrics, to send them to the mockup facility.

“In recent years the manufacturing industry has become increasingly frustrated by the fact that, despite an impressive level of superficially successful STEP model exchanges, it is usually extremely difficult to use a transferred model for any application that requires it to be modified in the receiving system.” – [Pra04]

The Parametrics Group in Working Group 12 of ISO TC184/SC4 [TC1] is trying to extend the STEP exchange standard to incorporate parametrics. The approach pursued is to require a set of basic modeling tools, features, and constraints, to exist in every modeling tool, so that a minimum of functionality can always be transferred to the target system. The problem, though, is that the ‘same’ sophisticated modeling tools and features behave slightly differently on the various systems. This concerns in particular the persistence of selections. In order to facilitate a correct interpretation of a selection in the receiving system, the procedural parametric description of an intelligent part is transmitted together with the ‘dumb’ instantiation as a mesh (B-rep). This way the receiving system can compare its own interpretation of the parametrics with the interpretation of the source system. Pratt comments on this approach as follows:

“The method described was proposed by a translator developer. It is admittedly not elegant, but is reported to be more reliable than any other approach that has been tried. It is possible that it will not meet all future needs encountered in the transfer of procedural models, but experience shows that it certainly handles simple cases.” – [Pra04]

Quest3D	www.quest3d.com	Halflife	www.valvesoftware.com
Virttools	www.virttools.com	Quake	www.idsoftware.com
Right Hemisphere	www.righthemisphere.com	Unreal	www.epicgames.com
Director	www.macromedia.com	Farcry	www.crytek.com

Figure 1.8: 3D Presentation software and commercial game engines.

Even without the complications through parametrics the exchange of CAD data between different software packages is a delicate issue. Since the programs are simply working differently it is often not clear, e.g., which tolerances correspond to each other in the source and target systems. The tolerance of a curve is, after all, determined by the numerical properties of the evaluation algorithm. Every NURBS curve, for instance, can equally be represented by a sequence of Bézier curves, at least in theory. But will both curves also have the same tolerances? – One might argue that this is a matter of programs conforming to standards. But STEP as well as IGES are very blown up standards since they had to represent the features of the different high-end CAD programs. Consequently many model im-/exporters implement them only partially.

The difficulties of model conversion have in any case stimulated a new lucrative market for specialized CAD conversion tools. Another solution, often pursued by bigger companies, is to simply acquire licenses of *all* major CAD packages. Due to the outrageous costs this is not a feasible solution for smaller companies. Autodesk, the vendor of AutoCAD, has reported a turnover of more than one billion dollars in 2004 [Vog05].

1.2.4 Online Rendering: Interactive 3D

As mentioned before there are two fundamentally different ways to present computer-generated imagery, pre-rendered films and interactive visualization. The advantage of films is that the author, the director, has complete control about what the audience sees, and about the moment when it is seen. Takes can be shot from perfectly chosen viewpoints, it is possible to show animated processes with excellent timing and rehearsal. The audio and video tracks are in perfect sync, so that, e.g., the spoken explanation in a documentary fits with the images that are shown.

The advantage of interactive presentations. Every user, also called *visitor*, of a virtual 3D world can choose the depth and length of explanations according to personal preferences. Interactive presentations can be more engaging since by definition they keep the level of involvement higher; at the risk, of course, that technicalities distract from the content. Interactive 3D stimulates the play instinct, and users typically try to explore the 3D world systematically, to discover all the possibilities for triggering actions, and to find out about the available interaction modes.

The downside is that authoring of good interactive presentations is very demanding and also somewhat involved and tedious. In principle every possible aspect of the interaction must be anticipated by the author, so that the system can react appropriately in response to the visitor's action. Interactive 3D is interesting only if the author has defined enough things to discover. This is also related to the issue of *interactive storytelling*.

The technical requirements for realtime 3D are demanding as well. With 20 fps the computer has less than 50 milliseconds to generate one image. This is usually possible only with decent pre-processing of the 3D data which involves, e.g., pre-computing the global illumination, hierarchical space partitioning, definition of cells and portals, and the reduction of the polygon count of the 3D models. Newer modeling tools also permit for *low-poly modeling*. It is important to keep in mind that, unless the viewer motion is explicitly restricted, a 3D model can be interactively explored from all possible sides. A model for offline rendering, though, needs to be detailed only where it is visible, very much like a film set.

The next question is which software environment to choose for presenting the 3D content. Possible options include:

- Commercial software for interactive 3D presentations
- Standard 3D viewers for standard 3D formats, e.g., for Web3D
- Abusing commercial game engines
- 3D Engines, open source or commercial, available as applications or as software libraries

Commercial 3D presentation software. Pre-fabricated solutions for the professional presentation of virtual worlds do exist. They are mainly targeted at (a) product design and (b) VR training applications. They do not include a modeler but permit to import externally created models into the VR authoring application, providing export plugins for the major modelers. The VR software distinguishes between the static environment and dynamic objects that may move in the scene. The motion of dynamic objects may be pre-defined (animations), based on rules ('artificial intelligence') for ambient motion such as vegetation or swarms and flocks of animals or crowd simulations, or triggered by events and visitor actions, of course ('behaviour'). These actions and the responses to them are defined in the VR authoring application typically by creating a *behaviour graph*, basically a network of event callbacks, sometimes combined with a state machine.

Authoring tools are optimized for productivity. Great efforts were taken to avoid VR authors typing in literal program source code. Following an 80:20 approach a good-looking, fluent virtual world with standard behaviour is created with a few mouse clicks. But only to walk about in a virtual 3D environment is quickly considered boring by most visitors. Interesting VR requires an engaging plot with a more complicated story book and, thus, a complex event network. Examples of state-of-the-art 3D presentation software are Virtools and Quest3D, listed in Fig. 1.8. They permit to create interactive 3D worlds on the level of small computer games without programming.

One fundamental problem is *sustainability*: Creating interesting 3D animations is very cost intensive because of the reasons already mentioned. An animation created with Virtools, however, can only be played in Virtools. There is no exchange standard for responsive 3D worlds, not even to speak of issues such as long-time archival. Transience is a serious problem. This is indicated, e.g., by the fact that the preferred way to deploy content created with Quest3D is as a *binary computer program* that can be executed (only) on current versions of MS Windows.

VRML as a VR exchange standard for Web3D. The general interoperability of interactive 3D was enthusiastically expected from the *Virtual Reality Modeling Language* (VRML) file format standard [VRM97]. With the rise of the internet during the 1990's and the new possibilities for information exchange through the world wide web a 3D internet, Web3D, was deemed to be in reach. Unfortunately, it turned out that the VRML approach from 1997 was maybe a little bit too naïve, for a number of reasons, some of which are:

- VRML is very restricted with respect to low-level shape representations; it does not have NURBS or Bézier curves or surfaces, or other free-form patches. It is too sloppy in terms of precision to be used as a CAD standard.
- In one way or the other every general 3D exchange standard has to cope with the problems mentioned before that IGES and STEP have to struggle with. In particular this concerns the issues of transmitting high-level shape dependencies and procedural information about the shape construction process, as well as annotations (*3D markup*).
- Despite its name VRML is not a modeling language. The modeling functionality in VRML is restricted to sweeping a polygonal profile along a polygonal axis (Extrusion), then there are heightfields and indexed geometry. But there is no viable way to change the connectivity of a mesh that is stored as an indexed face set (IFS) in a VRML file. For VRML, these modeling primitives are opaque objects, i.e., their internals cannot be accessed. Data access is limited to the pre-defined *fields* of each node type, such as the width and height fields of the Cylinder geometry node.
- Interactive VR requires three different languages to (a) define the VRML world, (b) define events by programming suitable callback functions in VRMLScript or JavaScript, and (c) possibly expand the VRMLscript functionality using PROTOs through the *External Authoring Interface* (EAI). Extensions are inevitable for more serious applications that require, e.g., a connection to a database to place an order. All source files for (a), (b), and (c) must be consistent, and they need to be kept in sync during development, which is extremely costly.
- The VRML standard comprises the scene description language but not a run-time environment. Similar as with HTML, the problem is that different VRML browsers display the content in different ways. Authoring a consistent VR world with VRML+Scripts+EAI-Extensions is a big investment, but a decent quality and a defined behaviour on the client side can still not be guaranteed. It depends on the quality of the VRML browser installed there.
- The state of the VRML scene graph is not part of the standard. Even if the ensemble VRML+Script permits the interactive manipulation of the scene graph, there is no standard way of saving the result of the manipulation as, e.g., an updated scene graph. This is disastrous for all commercial applications where the client expresses his wishes by manipulating a 3D scene, as in a *furniture configurator*, etc.
- VRML is a dead end: There is a strict separation between the VRML authoring application on the one hand and the VRML viewer on the other. Even worse, the authoring tool for VRML, the *world builder*, restricts itself to importing externally created meshes. This ruins any possibility to ever realize a feed-back from the VR world, that is interactively explored on the client side, back to the shape creation process, that takes place way before in the modeling application.

The successor of VRML is *Extensible 3D*, X3D, also administrated by the Web3D consortium [X3D04]. X3D is basically VRML in XML syntax, which remedies another drawback of VRML, its complicated syntax that lead to fragile parsers. XML is also the bridge from X3D to other standards from the world wide web consortium (W3C), such as SVG for 2D drawings [SVG03]. Despite its re-design and modularization X3D pursues the same fundamental approach as VRML and, thus, inherits most of its problems. The advantage of X3D over VRML was considered so negligible that still today many tools have a VRML export but not exporter to X3D. The fact that VRML is a *dead end* is also a feature that can be exploited for knowledge protection: A tessellated shape stored as IFS in VRML can not easily be used for any serious further shape processing without applying a bit of reverse engineering. – The wider adoption of X3D appears to be accelerating only recently, as 3D visualization is used more and more in industry for *downstream applications*.

Newer 3D file formats for downstream applications. In 2000 the following proprietary approaches for internet-based 3D visualization existed¹. Soon after 2000, when the internet hype was followed by disillusionment, some of the companies disappeared, and others got merged and realized new business models that are less hyped but more sound. In any case it is interesting to consider the question why these companies, and Web3D in general, failed to be a great success:

- **Metastream** is a proprietary compressed file format for the progressive transmission of high-quality triangle models rather than complete scenes. Comprises model encoder and free downloadable viewer for product visualization.
- **Blaxxun Contact** is a tool for *multi-user virtual worlds* with member accounts, chat, message boards, avatar support, object ownership and trading, etc. The business idea was to create *3D shopping malls* where customers could walk about. The *BS contact*, one of the standard VRML viewers, now belongs to *bitmanagement*.
- **Nemo** is a VR authoring software which introduced the concept of behaviour graphs, with hundreds of pre-defined behaviours that can be connected with routes to define the response to events without programming.
- **Parallelgraphics** offers the *Cortona* viewer with a number of much-needed extensions to VMRL, such as splines and NURBS. The proprietary authoring application offers advanced scripting capabilities for *3D user manuals*.
- **Superscape** offered the *e-Visualizer* as a very good-looking software renderer implemented in Java. Its great advantage is that it requires no installation procedure on the client side, and it works on any platform. The file format is proprietary and it contains, e.g., subdivision surfaces for curved surface parts.

To orbit around a single consumer product or to visit a 3D world without any purpose or challenge is apparently not very attractive. Visually and semantically richer, i.e., more expensive, VR can only be justified when it brings a concrete benefit, such as in VR-based training, where students can not spoil expensive machinery (*flight simulator*), and design reviews in industry or architecture: To build a real prototype for a design review is even more expensive than the visualization model, and it takes longer to build.

The new trend is to exploit and re-use the original CAD models as a source to derive information for the other product-related stages, called *downstream applications*: Printed manuals with automatically generated non-photorealistic illustrations and exploded views, interactive animations for assembly and disassembly, maintenance training lessons, archival, product history, feeding information systems for facility management in architecture, emergency plans, etc. Consequently two of the newer 3D file format initiatives to support downstream applications and interactive visualization were initiated by the CAD industry.

The first initiative is *JT Open*, released by the JT Open consortium under the leadership of Unigraphics, now UGS [JTO04]. It provides a downloadable interactive viewer that integrates also into office applications. Despite its name the initiative is not open, and technical specifications are only available to members. The special feature is that JT Open apparently permits to store also parametric information such as part/sub-part relations etc. The other initiative, *Universal 3D* (U3D), comes from Intel, RightHemisphere, Adobe, and others. It permits to integrate 3D data directly into .pdf documents [U3D04], the viewer is integrated with the Acrobat Reader. The U3D specification, which is publicly available, contains also optimized rendering primitives called CLOD-meshes (for *continuous level-of-detail*, see section 4.1.6).

Besides visualization both formats appear to retain the option to attach higher-level, parametric information to the entities in the scene. This flexibility permits to provide also future downstream applications with appropriate annotated data. It is doubtful, though, whether either of the formats will mature to the point where they permit to exchange intelligent parts between CAD systems. This would imply to solve the problems which STEP is working on since quite some time.

Commercial game engines. Only with the additional thrill of killing virtual enemies 3D has had its breakthrough on the mass market. Collaborative virtual worlds have become extremely popular in the gaming scene. The most distinct expression of the new sub-culture are *LAN-parties* where dozens or hundreds of *gamers* meet for a weekend to form combat groups fighting against each other in cyberspace with *ego-shooter* types of games. Today computer games are the '*killer application*' that drives the further development of 3D hardware and software. Games are marking the high-end of interactive 3D to the point that high-end graphics workstations are using today graphics hardware that was originally developed for gaming. The leading handful of cutting-edge computer games (see Fig. 1.8, right) are published primarily to promote the respective underlying *game engines*, which are the main source of revenues for these companies.

Remarkably, the static environment in most games is so static that even with the biggest guns it is not possible to shoot holes through the walls. The reason is that games heavily rely on pre-processed optimizations. Computing the global illumination, hierarchical space partitioning, cell-and-portal visibility, analyzing critical paths, etc., is all very time consuming, but it is indispensable for maintaining the mandatory 20 fps. – The revolutionary innovation of *Half-life II*, released in 2004/2005, is that for the first time it is possible to heavily affect the static environment in an ego-shooter.

There is a trade-off between flexibility and quality: The changes to the game scenery may not invalidate the costly pre-processing. But a changeable game world is a technical ingredient that can make a game much more interesting, and

¹presented in a tutorial on *3D on the Web* on the WWW9 conference in Amsterdam in 2000, together with Leif Kobbelt and Wolfgang Heidrich

Coin	clone of SGI OpenInventor	Irrlicht	game engine, shader support
OpenSG	scene graph engine, cluster support	Crystal Space	game engine, shader support
OpenSceneGraph	scene graph engine	OGRE	game engine, many features
Genesis 3D	WildTangent open source release	jMonkey Engine	game engine in Java

Figure 1.9: Popular open source 3D engines, general purpose as well as for games.

his greater influence on the world makes the player virtually feel more powerful, which leads to greater engagement. – Examples of this principle include popular games from the simulation genre:

- Parameterized car configurators make **racing games** more appealing with individually styled cars,
- the game **The Sims** permits to build up the every-day environment for a family of simulated characters,
- in the **Tycoon** series it is possible to put together railroad networks, or rollercoasters, and
- in **SimCity** the player can influence the evolving urbanization of a simulated developing society by creating streets, buildings, infrastructure, and everything that makes up a complete city.

Changeability is the biggest asset of digital 3D. The complete *changeability* of all three-dimensional entities, may they represent solid objects or animated characters, vegetation, furniture or industrial CAD datasets, is the greatest advantage of the digital *cyberspace* and the *virtual reality* produced by a computer. This fact has been noted before [Gib84, the99], but current software approaches, from computer games to Web3D, still do not realize this potential sufficiently.

The central question is: What is the meaning of *changeability* with respect to three-dimensional shapes? Better understanding of high-level shape descriptions seems necessary in order to make descriptions more easily changeable.

The most advanced concepts available today for creating, editing, and improving shape are those in parametric CAD systems for professional industrial design. To increase the level of changeability in computer games it may therefore be advisable to integrate parametrics into 3D games. Parametrics open the door to powerful high-level changes to simulated geometry. But in computer games usability is key, and parametrics for games must be different from those for CAD. It is very inspiring to imagine in which ways parametric shape design might have to be adapted in order to make it usable for games: Tools and parameters must be obvious, intuitive, responsive, and not too many of them may be offered at a time. Shape modeling can probably benefit also drastically by adopting principles from game design. Furthermore the availability of full shape modeling capabilities in a game engine would pose interesting technical challenges as it would require partial updates of the pre-processed data, and to selectively redo the whole pre-processing pipeline at runtime.

3D Engines. The number of available 3D engines is absolutely amazing. The *3D engines database* alone lists around 200 of them, commercial and free [Dev], see Fig. 1.9. Around 900 open source projects whose description contains the keyword *3D rendering* are hosted on *sourceforge*, and 400 of them contain also *3D modeling* [Sou]. The different projects vary greatly in purpose as well as in degree of maturity and developing activity. Some engines exist that can be accessed on a high level using a scripting language, and others are only software libraries to facilitate the development of applications on top of the two principal low-level 3D graphics APIs, DirectX and OpenGL [Dir, WND97].

Despite this apparent variety all the approaches share the same fundamental problem, the separation of modeling from rendering. There are open source modeling packages like Blender or OpenCascade on the one hand, and dedicated rendering engines like OpenSG or Java3D on the other. They offer a number of importers for the most common formats; imported models are treated as anonymous objects, only with a bounding box, that can be inserted into a scene graph.

No sustainable file formats and no interoperable 3D software. It is an interesting question why, despite all the available tools, 3D is still not used as much as it could; why, for instance, not every software that has to do with real-world items can also show the location of these items in their respective contexts. Why is not every storekeeping application provided with an abstracted 3D interface to the individual shelves? When this is a productive means to keep track of huge virtual stores in a computer game, why should it not help in practice? – A diligent review of various software applications might very well reveal a huge number of options where 3D could greatly increase productivity.

Why not simply use the solid modeling library *OpenCascade* together with the scenegraph engine *OpenSG* to provide business and office applications with a 3D visualization? Because there is no single such combination that fits for all purposes. Two key properties are characteristic for 3D:

- The applicability of modeling+rendering for a great variety of different purposes, and
- the fact that interactive 3D resembles more executing an application, than playing an audio or video sequence.

The fractionization of 3D approaches, positively pluralistic, has become an obstacle today because the link between them is missing: There is no general way to exchange interactive scenes, and the different ‘3D-players’ are not inter-operable.

1.3 Modeling in Computer Graphics

The term *modeling* is used in computer graphics research not only for techniques concerning interactive shape design. Articles from three different research areas are labeled with the same keyword ‘modeling’:

- **shape representations** are low-level methods to store surfaces or volumes in a digital computer,
- **shape processing** creates, converts between, and improves on these low-level representations, and
- **shape design** is on new ways for authoring or retouching shape more or less interactively.

Two other highly active areas of modeling are **animation** and **simulation**. They are less relevant for this thesis, though.

1.3.1 Low-Level Shape Representations: Lists of Primitives

In computer graphics, three-dimensional solid objects are usually described only in terms of their surface. The surface is sufficient for rendering images as long as the light cannot enter the object, i.e., the material is not transparent or translucent (like marble). For this reason a great number of different methods have been devised to represent in a digital computer two-dimensional surfaces embedded in three-space. It is important to note that on the lowest level shape is almost always represented by enumerating some sort of geometric shape constituents, or atomic shape elements. They are subsequently referred to as *geometric primitives*, and will now be shortly described.

The following list gives a rough impression of the variety of common low-level shape representations. It is not meant to be exhaustive or as a complete taxonomy, but just to highlight the kinds of data structures and operations that are typically employed. They are also at the heart of all commercial, procedural or parametric, modeling systems.

- **Constructive Solid Geometry (CSG)**

Parameterized solids such as box, cone, sphere, cylinder, etc. are the leaves of the *CSG tree*. Its inner nodes represent *Boolean set operations*: union, intersection, difference. This corresponds nicely with the objects and operations used for building machines in classical engineering. CSG was enthusiastically employed since the 1970s, until more and more freeform shapes were used in engineering, which are difficult to represent with finite CSG trees.

A CSG object can be represented in a straightforward way by a binary tree. It is built by setting primitives into the scene. Successively a pair of objects is selected and a set operation is applied to combine them into one.

- **Boundary representations (B-reps)**

The closed surface of a solid, its boundary, can be partitioned into segments, the *surface patches*. Adjacent patches meet along a common border curve, which induces on the surface the structure of an abstract graph that is locally planar: the *B-rep*. Its vertices, edges, and faces have an attached embedding that maps them to 0D points, 1D curves, and 2D patches embedded in three-space. The abstract B-rep can in principle be combined with any method to represent patches and border curves, as long as it permits to keep the surface consistent (closed, orientable, ...).

The B-Rep graph is represented by enumerating the connectivity of vertices, edges, and faces, i.e., the *incidence relation*. Sets of operators exist that permit to build up a B-rep successively, and to apply local modifications to the graph as well as to its embedding. A B-rep can also be created by converting from, e.g., a CSG tree. Different possibilities exist to represent the geometric embedding (patches). This is explained in detail in chapter 2.

- **Parametric patches: Splines, NURBS, Bézier tensor product surfaces, Hermite-, Coons patches, etc.**

Parametric freeform patches are smooth mappings $\mathbb{R}^2 \rightarrow \mathbb{R}^3$, which makes it simple to generate explicit surface points, but difficult to test whether a given point in \mathbb{R}^3 belongs to the surface. The fundamentals of representing parametric surface patches date back to the 1960’s and 1970’s. As the surface of most objects consists of more than one patch it is most important to assert that adjacent patches meet smoothly along their borders. This was the reason for developing the theory of *geometric continuity* (see the book from Gerald Farin [Far02]).

Smooth patches are typically represented by a couple of control points (*control vertices*, CVs). They use to be arranged in a regular grid, triangular or quadrilateral (*tensor-product patches*). The CV grid is a convenient fashion of storing the coefficients of piecewise polynomial (or rational) functions. Basic operations are to split a patch in two, to stitch two patches together, and to add a row or column of CVs using *degree elevation*.

- **Triangle meshes (simplicial complexes, triangle soups)**

The simplest patch type is a linear surface, a plane, as specified by three points in \mathbb{R}^3 . Since triangles are so simple, more of them are needed, usually many more. But any type of surface can be converted to a *triangle mesh* simply by *sampling* and connecting nearby sample points. A triangle mesh is typically represented by a list of vertices and a list of index triplets, one for each triangle. A set of triangles floating in space, a *triangle soup*, is the ‘smallest common denominator’ of a surface. Triangle meshes are the dominant surface representation today because

- the graphics hardware is optimized for rendering triangles at high speed, and
- shape acquisition, 3D scanning, usually produces dense triangle meshes.

Starting from a generic *base mesh* (tetrahedron), a triangle mesh can also be built up iteratively: A *vertex split* applied to a vertex turns it to a pair of vertices connected by an edge, unfolding one triangle on either side of it.

- **Point clouds**

Points are gradually superseding triangles as the dominant low-level shape representation. They are even more basic and simpler to create than triangles. The resolution of a point cloud can be adjusted by simply “forgetting” some of the points, without the overhead of maintaining triangle connectivity. The original work from Pfister, who found a way to render point clouds efficiently [PG04], has triggered enormous research interest, and a wealth of efficient techniques for storing, processing, transmitting, and rendering point clouds have been developed [PG04, ADG*03].

A point cloud is represented by a long list of coordinate triplets. The surface normals are not stored explicitly but derived from the cloud by *principal component analysis* (PCA) of the local covariance matrix of the surface points.

- **Implicit surfaces**

An implicit surface is defined by a function $\mathbb{R}^3 \rightarrow \mathbb{R}$ that permits to check whether a point belongs to the surface. The surface is defined as the *zero-set* of the function. Every surface can be converted to implicit form, for instance as the *signed distance function*: It is > 0 above and < 0 below the surface. Generating explicit points on the surface is more difficult than with parametric patches, but it is usually simpler to intersect a ray with a surface when it is in implicit form (e.g., a sphere).

Implicit representations include *metaballs*, points with a spherical exponential power field blending close spheres together, and also the *blend tree* where, similar to a CSG tree, implicit primitives in the leafs are successively combined by various possible blending functions [BBB*97]. Many other implicit representations exist, just to mention *radial basis functions*, the *partition of unity implicits*, and *skin surfaces* [CBC*01, OBA*03, CDES01]

- **Volumetric Models**

Similar to implicit functions, volumetric objects are defined by a *density function* $\mathbb{R}^3 \rightarrow \mathbb{R}$. It can be interpreted as the *opacity* of the volume, like the density of tissue recorded by computer tomography. The regular sampling of a volumetric function is memory intensive: A subdivided cube with $6n^2$ quads on its surface contains n^3 volume elements (voxels) in its interior. Density values can also be mapped to different colors. To render a volume requires for each pixel to integrate this color/density along the ray through the volume. With programmable graphics hardware this is possible at interactive rates [EKE01]. Hierarchical or irregular (tetrahedral) volume meshes can be more memory efficient with variably varying data. They can be ray traced using graphics hardware as well [WMKE04].

Besides regular or irregular sampling, volumetric models can also be synthesized by combining implicit/density functions. A notable example is the *F-rep* framework that offers a number of volumetric primitives, for instance trivariate (volumetric) patches [PA04, SPS04], that can be combined in different ways, e.g., using set operations.

- **Multi-resolution surfaces, progressive meshes, and subdivision surfaces**

The shape representations presented so far all consist of certain geometric primitives together with shape operations to manipulate the primitives. The operations may also be applied in a very schematic way: A quadrangle is replaced by four smaller quads simply by inserting and connecting a vertex on every edge and face. Similarly a triangle can be replaced by four smaller triangles. Schematic repetition of refinement operations permits to predict the structure of the result; when suitably defined, a smooth surface results in the limit. – By re-sampling a given triangle mesh, the triangles can be piecewise re-arranged to resemble a multiply subdivided mesh. This *subdivision connectivity* then permits to “undo” the refinement, i.e., to coarsen the surface, and to refine it again when needed.

But note that the operations of *any* shape representation can be applied schematically. In a similar fashion, it is possible to define multiresolution CGS trees, B-reps, patches, point clouds, implicits, or volumes. Such generalized multiresolution shapes are sometimes also called *progressive*.

- **VRML as an example of a scene graph**

A *scene graph* is a tree whose nodes are shapes or affine transformations. Further *node types* are material, camera, light source, background texture, and also behaviour related types such as sensor and interpolator. A node is defined by a number of *fields*, whose values can be a single number, a string, 3D vector, node, etc., or a whole array, such as the vertex positions and indices of an IFS or the children of a transformation. By using DEF/ROUTE the value of a timer can be routed to an interpolator node that smoothly and periodically interpolates between two sets of 3D positions of an indexed face set representing, e.g., a swimming fish. With DEF/USE the same node can be inserted multiple times in the scene graph (reference objects), which turns it effectively into a DAG.

Shape descriptions: Objects versus operations. One way to look at shape is the *database view*: Any given 3D object, in either of the shape representations mentioned above, could easily be stored in a relational database. Every type of geometric primitive, point, triangle, or NURBS patch, can be mapped to a database record containing only a few Booleans, integers, floats, and strings. A list of such records, a database table, can represent any 3D object then. – Unlike the record entries in an address database, however, entries in the records of a shape database are usually not independent. Entries in different records are related by rules, but these rules are not represented in the database: A cloud of points on a sphere or a number of coplanar triangles can be represented much more efficiently.

The last example, VRML, illustrates that the (hierarchical) database concept is indeed used in computer graphics. VRML goes beyond a simple relational table, though: The DEF/ROUTE concept permits to relate fields of different nodes by expressions, so that dependencies can be explicitly formulated. But unfortunately, VRML is inconsequential in that it does not permit to express those dependencies that are needed so much: *shape dependencies*.

A solution comes from an alternative view that emphasizes the importance of the *shape operations* over the data records. This is illustrated by the example of multiresolution surfaces which, with enormous success, replace data by sequences of operations. A multi-resolution surface is not a static piece of data; it is data plus an algorithm to coarsen or refine the surface resolution. This *dynamification* of static data was the catalysator for a whole class of new algorithms to analyse and process shapes.

1.3.2 Shape Processing

The variety of existing shape representations from the last chapter suggests that each of them has its strengths and weaknesses. The representation is chosen as a function of the data available, their purpose, and the available algorithms. Furthermore all kinds of conversions are possible, and needed, from the triangulation of (implicit) iso-surfaces to the local implicitization of point clouds. This leads to the very hot research area of *geometry processing*. It is just as rich as it is interesting as, e.g., documented in the proceeding of the *Symposium on Geometry Processing* that was held 2003 in Aachen, Germany:

- accurate *re-sampling* with guaranteed Hausdorff distance, or with mesh regularization [\[BO03, SG03\]](#)
- *signal processing* on surfaces, multiresolution analysis, and mesh encoding [\[BM03, SCOT03\]](#)
- *data structures* for non-manifold meshes and abstract simplicial complexes [\[FH03\]](#)
- *recovering features*: repairing scanning artefacts on chamfered sharp edges (creases) [\[AFRS03\]](#)
- *point clouds*: statistical rendering, and ray intersection using a local distance field [\[KV03, AA03\]](#)
- *conformal surface parametrizations* that preserve angles and scale distances [\[GY03\]](#)
- *voxelization* by computing the distance field with respect to the max-norm (l_∞ -norm) [\[VKK*03\]](#)
- *mesh repair* by hole filling, re-triangulation, and fairing [\[Lie03\]](#)
- *alternative surface representations*: geometry images as a form of shape textures [\[LHSW03, SWG*03\]](#)

The use of shape features in geometry processing. It is important to realize that all methods for shape processing implicitly speculate about the ‘true shape’ of an object. Some of the methods are solely based on phenomena (‘most objects exhibit flat surface regions’), and others are technology driven: *Shape features* are those shape elements for which there is a chance to recognize them automatically, like flat parts, sharp creases, regions with constant curvature, etc. But things like *symmetry* or *linear sequence* can be features of a shape as well; but since they are more abstract they are much more difficult to find and, when found, more difficult to store.

The difficulties of shape recognition in general have been mentioned in the context of shape matching. One area where feature recognition is applied very successfully, however, is *reverse engineering*. It uses a feature extraction process that partitions the shape into surface segments. These features are then mapped to corresponding features in a parametric modeling system, and then the frozen scanned 3D object gains its life back: Holes for bolts can be slightly moved, the radius of the bevel can be adjusted, etc. [\[Kre00\]](#). The fact that reverse engineering works well for industrial engineering is obviously tightly related to the fact that this domain uses only the aforementioned limited set of modeling operations. But note that, still, parametrics capture only part of the design intent: To recognize a ‘bevel’ feature in a mesh is fine, but it would even be greater to find out that two different bevels have the same radius.

Signal processing on surfaces. A point of view that can be very useful for mesh processing is to understand a surface as a two-dimensional signal. The 2D signal processing approach is especially successful for image processing, where the discrete cosine transform (DCT) is an efficient device for the transformation of an image into the frequency domain. In frequency space much higher compression ratios can be achieved with respect to the same visual degradation [\[jpe00\]](#). Also local retouche tools such as Gaussian smoothing, edge detection, and embossing are based on signal processing. Signal theory was also the theoretical device that enabled the great success of digital audio and the *compact disc* (CD). Fast *digital signal processing* (DSP) chips permit to apply all kinds of audio effects, from flanger and ... to coding/decoding.

The signal processing view has also its limits, though. This is revealed just by the analogy between shape and images or audio: There is a difference between *music* and an *audio signal*, which is very much like the difference between *shape* and a *surface signal*. Music is not composed by enumerating audio samples; there is an unlimited number of different arrangements, or *cover versions*, of the ‘same’ song. Both aspects are also reflected on the technical level, as the difference between MIDI signals and sound samples: The audio signal is a complex mixture of the sound from different musical instruments, and each of the instruments is steered by a stream of musical notes arriving (in case of electronic instruments) over the MIDI bus. And it is just this ensemble of all notes for all instruments, the musical score, written down by a composer, which determines the identity of a piece of music.

From music back to shape, signal processing on geometry permits to apply retouche tools and editing effects to a shape. But this does not reveal the shape either: Effects are applied only to improve an already existing signal. Shape is created physically using tools like hammer and chisel, saw and drill, welding and casting. The analogon to MIDI on shapes would be the commands sent to a digital shape tool, e.g., a CNC milling machine. It is therefore interesting to see which shape editing tools have been developed.

1.3.3 Interactive Shape Design

The previous subsections have provided the prerequisites for designing shape interactively: Methods to store volumes and surfaces in a digital computer, and tools to operate on and to process, i.e., manipulate, these surfaces. From a formal point of view sculpting only means to edit the geometric and topological relations between shape primitives. But from an artistic point of view all that is perceived is the editing operations; the details of the underlying low-level shape representation are completely irrelevant for the artist. Interesting is only which kinds of editing operations it supports, and the features and properties of the resulting surfaces.

Interactive shape design therefore implies a change of perspective: Not the representation of shape is important but what can be done with it. The emphasis on shape *operations* is an immediate consequence of design as a process where a designer is searching her way through *design space*, i.e., the space of all possible solution shapes. – This is of course directly related to the shape description problem from the introduction: Which sort of operations would permit to go very straight the way from problem to solution in design space, for a particular class of shapes?

In the following a rough overview will be given over the variety of existing techniques for shape design. The main reason for this variety is, of course, that all such approaches need to integrate three different things: A shape representation, shape operations, and a design metaphor. This makes for a great number of possible combinations.

Volumetric sculpting and digital clay. The fundamental dichotomy between the continuous and the discrete is part of the history of shape design from its very beginning: Probably even the first humans have used both stones and clay to give shape to their ideas. In any case, a piece of clay is perceived as one of the most basic methods for creating shape.

A digital clay metaphor is the idea behind the *Kizamu* system [PF01]. It uses an *adaptively sampled distance field* (ADF), essentially a large octree (up to level 10) of distance values. They can be integrated to obtain a continuous distance field to a surface. The distance function permits, e.g., a sculpting tool to follow the surface in a prescribed distance. Tool paths are Bézier curves, which can therefore be edited, and also scripted, and re-played in high accuracy. – The ADF very much smoothens out sharp edges of meshes converted to this volumetric representation.

Marie-Paule Cani and her colleagues focus on practical volumetric sculpting in [FCG99], also using an implicit surface defined by a regular volumetric grid. Their emphasis is on the *sculpture metaphor* and the tools for sculpting: Material deposit with a virtual *toothpaste*, a (soft) eraser to remove material, a (soft) surface painter, and furthermore small objects for imprinting stamps on the surface. To support design exploration, 200 undo-steps are supported, by storing each time a snapshot of the complete model in a separate file: There is no reasonable set of invertible elementary operations.

Another volumetric grid approach is from Hong Qin and Jin Hua. The implicit surface is given through a trivariate scalar-valued B-spline $s(u, v, w) = \sum_{i=0}^{l-1} \sum_{j=0}^{m-1} \sum_{k=0}^{n-1} \alpha_{ijk} B_i(u) C_j(v) D_k(w)$ with a grid of control scalars α_{ijk} . The surface can be evaluated fast enough for haptic feedback with the high *haptic frame rate* of at least 1000 Hz. Sculpting on the level of individual volume cells with a PHANToM device becomes possible with this approach, on the price of heavy aliasing artifacts. The detailed editing operations resemble very much three-dimensional bitmap editing: A chiseling operation to locally add/remove one or two voxel layers (embossing), volumetric copy and paste, and inflation and deflation of one surface voxel layer. Moving and bending on volume cells can be thought of as ‘voxel switching macros’.

The great problem of regular volumetric grids is that aliasing artifacts are unavoidable, e.g., when converting a mesh to it. Axis-aligned cubic boxes are fine, but a regular pentagonal prism is a problem; worst are slight rotations of axis-aligned shapes. Aliasing can be alleviated by using a surface representation with built-in smoothing, which is the motivation for employing the *level set* method as did Museth et al. [MBWB02]. The surface is then the result of a partial differential equation (PDE) from a *speed function* defined on about $256^3 \approx 16$ M voxels. To reduce aliasing artifacts the implicit function is converted back to a mesh using marching cubes on a denser grid.

Alternative volumetric approaches. One way to avoid the aliasing issue is by a hybrid representation combining the precision of meshes and the blending power of implicit surfaces. The *surface flow* from Duan, Hua, and Qin [DHQed] embeds a mesh in a distance field and solves a local PDE iteratively for smoothing and blending shapes. The distance field nevertheless permits for the usual volumetric operations such as drilling, sketching, cutting, and even Boolean operations.

Consequent pursuit of this idea leads to directly warping the space that the model is embedded in. Chua and Neumann use in [CN00] a volumetric free-form deformation (FFD) with a grid of $4 \cdot 4 \cdot 4 = 64$ spline CVs that define a mapping $\mathbb{R}^3 \rightarrow \mathbb{R}^3$. They also propose a hardware-supported evaluation of the spatial deformation, which was not yet possible in 2000, but should be no problem today with the programmable vertex and pixel shaders.

Another variation of this idea is to use only local space warps for shape modeling. The ingenious *Twister* approach from Llamas, Kim, Rossignac et al. [LKG*03] introduces one single well-defined operation that can simultaneously translate, twist (screw), and bend a shape. It has two very intuitive parameters, namely a start and an end coordinate frame plus a radius of influence; the latter defines the extend of the space deformation leading from one frame to the other. Plausibility and predictability are considered more important than, e.g., physical correctness. Interactively executed warps can be logged, scripted, and replayed offline in higher precision. As input device one – or two, for a two-handed twist – Polhemus trackers are used for a grab-and-drag metaphor with a moving tripod gizmo.

An alternative to warping space is to use volumetric primitives. The idea of Pizer and Thall is to reverse the process of *skeletonization* and to use it for modeling [PT00, TPF00]. The *medial axis* of a solid object is the locus of the midpoints of all maximal spheres inscribed in the object (see section 4.5.5). The *M-rep* representation employs essentially this skeleton for shape modeling, attributing a sphere radius to each skeleton point. Supported medial axis primitives are quad grids, tubes, and slices plus distance to the boundary.

A simpler set of volumetric primitives is introduced by Edelsbrunner and Cheng, namely *skin surfaces* [Ede99, CDES01], superficially similar to Blinn’s metaballs [Bli82]. A skin surface is specified through a finite set of weighted points (i.e., spheres) that are combined to a single surface by a tangent continuous implicit blend. A skin surface is free of self-intersections and maintains a consistent overall topology. Every closed surface can be approximated efficiently with a skin surface, although sharp edges (creases) pose a certain problem. The skin varies continuously with points and weights, which is a very desirable property. Efficient skin triangulation algorithms exist.

Particle- and point-based techniques. The classical article on direct shape manipulation with oriented particles is [WH94] from Witkin and Heckbert from 1994. The problem is that primitive shapes such as (implicit) super-quadratics are controlled by equations with un-intuitive parameters. Particles have a dual use here: They are generated for rendering the implicit shape, but they can also be used as gizmos for interactively pulling and pushing points on the surface to the desired form, for instance to adjust the ends of an implicitly defined cylinder.

This idea was extended in 2002 by Hart, Bachtá and others [HBJF02]. They note that moving a particle on the surface can have different meanings, e.g., to move or to scale the shape. They propose a set of *adapters*, each of which specifies a way of accommodating a position change of a surface particle by changing some parameters in the implicit equation. – They also note that ‘implicits are slippery’: pulling a particle is simpler than pushing it. In a subsequent article Su and Hart present a programmable particle system C++-framework as a ‘Renderman for particles’ [SH05]. They differentiate between attributes (data), behaviours (motion), and shader (rendering) of a particle type. Examples include particles that wander towards shape singularities, silhouette particles, meshing particles, and clustering particles.

The *Skin* approach from Markosian et al. [MCCH99] uses conventional polyhedral objects (spheres, boxes, cylinders, meshes) as skeletons. They guide the interactive growing of a combined particle/subdivision surface that approximately keeps at a certain distance from the (union of the) skeleton elements. Besides guiding the growth the user can adjust the offset distance from skeleton elements, and he may draw/edit crease curves in the surface.

The modern form of oriented particles are point clouds. A good overview of the techniques related to point clouds is given by Kobbelt and Botsch in [KB05]: The local covariance matrix from the k -nearest neighbours to determine the surface normal, elliptical and EWA splats for rendering, *moving least squares* (MLS) with a Gaussian filter kernel as a smooth local surface approximation, resampling and particle repulsion, volumetric FFD for sculpting, and many more.

Modeling operations for point clouds are exemplified in *Pointshop3D* from Zwicker et al. [ZPKG02]. The idea behind it is a *Photoshop for point clouds*, and correspondingly the central concept is a *brush*, a selected rectangular surface patch to work on. The operations are filtering, carving, normal displacements, direct surface painting, and applying a texture. The technical foundations are a fast surface resampling method and a ‘minimum distortion’ parametrization.

The idea to treat point clouds like bitmap images is further elaborated by Pauly et al. in [PKKG03b]. The MLS method defines a surface locally by fitting a reference plane that minimizes the sum of the squared distances from points in the cloud, weighted by a Gaussian kernel. This permits to (locally) derive a signed distance function to the surface and together with a kd-tree this even allows for closest point queries: Which surface point is closest to a given $p \in \mathbb{R}^3$? This, in turn, is the basis not only for collision detection and interactive Boolean operations, but also for all sorts of local deformation tools that can even resolve surface self-intersections. – Updating a kd-tree is a problem, though.

Simulations and physics-based sculpting. One strand of research is concerned with the idea that a faithful, or at least plausible, simulation of the physical reality would make for more intuitive shape modeling. In 1995 Hong Qin presented *dynamic nurbs* [QT95] as a free-form representation with built-in physical properties. Users are urged to formulate design intent as forces, explicitly incorporating time, so that the surface can converge to the desired shape. The idea was later extended to FEM-based dynamic subdivision [MQV98, MQV00]. The *finite element method* (FEM) was directly built in to interpolatory butterfly subdivision so that the surface deforms in an intuitive way when applying forces to CVs.

The *Dynasoar* approach, which stands for *DYNAMIC Solid Objects of ARbitrary topology*, was developed by Qin and McDonnell. It takes the idea further by integrating a mass-and-spring system with volumetric Catmull/Clark subdivision solids [MQW01, MQ02, McD03]. The problem with direct volumetric sculpting is the large number of degrees of freedom. Physical simulation (in principle) permits to change many DOFs at a time, and Qin and McDonnell argue that in some sense the ‘right’ parameters are exposed to the user. Modeling is done with a rope tool to pull surface points, but also with more conventional volumetric extrusion and cell removal with an option to fixate parts.

Subdivision surfaces furthermore offer the possibility to spread the simulation over multiple scales. Finite elements are combined with hierarchical multiresolution subdivision solids by Capell et al. in [CGC*02]. They integrate the kinetic and potential energy per volume cell to obtain elastic deformable trivariate solids of the McCracken/Joy (catmull/clark) type. They are interested on pure simulation (falling rubber duck), so there are no modeling operations as such.

A large body of research on deformable models is motivated by surgical simulations. Brown et al., for instance, use a direct (non-tetrahedral) volumetric mass-spring system to simulate viscoelasticity [BSB*01], together with a sphere tree to detect collisions of the deforming surfaces. The primary surgical modeling operation is to cut tissue, which requires resampling along the cut, and sewing tissue together. Both operations are supported by this approach. The precise target application is *microsurgery*, where very small instruments are manipulated indirectly. This implies that no force feedback needs to be simulated, which would be great hurdle due to the required high ‘haptic frame rate’. Haptic feedback is possible only with a consequent multi-resolution approach such as, e.g., presented by DeBunne et al. in [DDCB01]. They manage multiple quasi-uniform tetrahedral meshes, at different levels of resolution, which are quickly synthesized at the position where the surgery tool touches the tissue. The highly optimized volumetric meshing scheme unfortunately makes neighbourhood variations, i.e., cutting tissue, very complicated. They are not supported by this approach.

Editing fair triangle meshes and multiresolution surfaces. The classical article on variational sculpting with triangle meshes is [WW94] from Witkin and Welch in 1994. They developed a toolbox with basic techniques, e.g., for fairing the surface by minimizing the principal curvatures, a local parametrization scheme, and a resampling method to assert the quality of the triangulation. Modeling is done with external shape controllers, e.g., a sphere, or a cylindrical offset tool to create pipes. Topology changes, surface cut and merge, is done along curves embedded in the surface. Each modeling step is followed by a fairing step to maintain a fair surface quality and good triangle aspect ratios.

With the advent of multiresolution analysis and ‘signal processing on meshes’ it became possible to differentiate frequency bands on a surface. A natural way to exploit this for modeling leads to the concept of *multiresolution mesh modeling*. Major papers are [ZSS97] from Zorin et al. and [KCVS98] from Kobbelt et al. The latter permits to manipulate and deform a base mesh, and to re-apply the details from the simplification hierarchy, while the first uses smoothing and then subdivision to obtain a finer resolution to apply small-scale detail to. Such techniques were the basis for the *normal meshes* approach for patch-wise rendering of displaced surfaces from Guskov et al., and then to the representation of shape as *geometry images* from Gu, Gortler, and Hoppe [GVSS00, GGH02]. Both representations can be rendered efficiently using programmable graphics hardware.

In the context of modeling, the multiresolution idea was further explored by Kobbelt et al. in the ‘lava lamp’ approach [KBS00]. It uses a two-band representation where the low frequency part can be manipulated with a number of *control ellipsoid* over which a thin membrane is spanned. Its triangulation is identical to the triangulation of a simplified shape, to which the detail coefficient can be readily re-applied. This was further elaborated by Kobbelt and Botsch in [BK04] for fair multiresolution modeling with boundary constraints. A closed path in the mesh defines the region of interest (ROI). Within the ROI the mesh can be edited directly by pulling a handle, attached to a selectable handle region, with adjustable stiffness and boundary smoothness via sliders, and fullness/pointedness through the size of the handle region.

Another way to use multiresolution for modeling is by shape templates on different scales. In their article [LLS01] Litke, Levin, and Schroeder fit a low-resolution Catmull/Clark surface (standard mannequin head) to a high-resolution input mesh (laser-scan of a head) by quasi-interpolation, which yields a Catmull/Clark surface with detail coefficients. The correspondence allows to manipulate the high-res triangle mesh by manipulating the low-res subdivision control mesh. Applications are embossing, by changing detail coefficients only locally, and progressive transmission of the details while the smooth subdivision surface is already being displayed. In [BMBZ02] Biermann et al. have shown that in a similar fashion also cut + paste editing of multiresolution surfaces is possible. They point out the problem to determine *which* displacements from the source region constitute the actual feature: A carving on a curved surface should be flattened out, and a writing on rock is hard to distinguish from the noisy rock surface.

Sketch-based techniques. With respect to usability, technology-driven modeling approaches are very questionable, where the user is urged to create and manipulate shape by directly editing the DOFs of the low-level shape representation. A notable example are NURBS and the extensive CV-editing they require.

In the article [ZHH96] on their classical SKETCH approach, Zeleznik and his colleagues point out the “*dichotomy of how easy it is to depict a 3D object with just a pencil and paper, and how hard it is to model the same object using a multithousand dollar workstation.*”. Consequently, they propose to use multi-stroke gestures for modeling CSG-like assemblies of primitives such as cylinder, cube, etc. Placement constraints are to be suggested in the drawing so that the design intent can be inferred automatically by the computer. But ambiguities are still the main problem of this approach and, thus, Zeleznik and his colleagues propose to extend the set of basic gestures by additional context-sensitive gestures, whose effect depends on the object the cursor is over.

There is again a raising interest in the long-standing research domain of sketch-based techniques. This is documented, e.g., by the *First Eurographics Workshop on Sketch-Based Interfaces and Modeling 2004* in Grenoble [HPS04]. Varley et al., for example, pointed out there the fact that also “*CAD engineers write ideas on paper*”. CAD is good for the exact realization of an idea, but not for developing ideas in the first place. They also mention some classical problems of sketch-based methods: At junctions, which lines come nearer, which go farther? Which edges are supposed to be parallel in 3D? Which lines are convex/concave? Even if results are acceptable with agglomerations of CSG-like primitives, the general approach is difficult to extend so that it handles also curved free-form geometry. Varley et al. propose to first draw polyhedral objects, and then to draw freeform curves on top of them in a second stage [VTMH04].

The *Teddy* system from Igarashi et al. [IMT99] lets first-time users create free-form models within minutes. Its minimalistic user interface does not provide CVs or WIMP-style controls (for *Window, Icon, Menu, Pointer*). Instead it uses only multi-stroke gestures. Genus 0-objects are created by drawing their silhouette that is automatically inflated. The same way extrusions are added and cavities are dug. Parts can be cut away, edges may be sharpened or smoothed out, and the whole shape can be bent relative to a reference stroke. – The system from Karpenko et al. [KHR02] uses not a mesh like *Teddy* but inflates closed strokes in the image plane with variational implicits (‘blobs’). The depth of a blob can be adjusted by dragging its shadow on the ground. Implicit blobs can be smoothly blended while still maintaining a discrete ‘blob hierarchy’. This permits for hierarchical animations and for operations like mirroring a sub-graph (‘symmetrization’).

A reduction of a shape to a few essential pencil strokes is the goal of *non-photorealistic rendering* (NPR). Nealen et al. therefore note that shape sketching can be understood as ‘*inverse NPR*’ [NSACO05]. This suggests an interpretation of pencil strokes as a contour (silhouette), or as a feature line (crease) within the surface. Strokes can therefore be used in their system for a (multiresolution) deformation of the object silhouette (which acts as the reference stroke), to sketch a sharp crease, or to sketch a smooth ridge/ravine in the surface.

Surface drawing, tangible interfaces Deemed as even more intuitive than sketch-based modeling is direct drawing of shape in space. The hypothesis is that spatial manipulation enhances shape understanding. The *Surface Drawing* approach from Schkolne et al. [SPS01] uses see-through glasses, a cyberglove, and tangible devices with a free 3D surface painting metaphor: The hand is for drawing surface strokes and moving them, a pair of kitchen tongs for scaling, an eraser tool is of squeezable rubber, and a flexible magnet tool deforms. Extensive user studies have revealed that to some extent, precision modeling and intuitive handling are opposites: Obtaining precision requires more complicated tools. Drawing a straight line in free space is more difficult than on a table. – Wong, Lau, and Ma also employ a cyberglove and use it for surface editing with a magic B-spline control hand [WLM00]. The palm of the hand is modeled as a parametric patch which is then projected on an object, so that the hand deformation directly translates to an object deformation.

The *FreeDrawer* (Avango) toolkit from Wesche and Seidel [WS01] attempts at obtaining CAD-like precision by reducing the number of DOFs. The user can not draw surfaces but only a network of curves in space using a tracked stylus. The basic modeling operation is to attach a new curve to an existing curve, and to edit (smooth/sharpen/move) a curve within an adjustable 1d-ROI using a ‘flat-iron metaphor’. The curve network is interpolated with a ‘Kuriyama’ surface (n -sided blend), to which a Catmull/Clark surface is fitted. This assures a smooth transition between adjacent patches. Tool selection is done with an innovative arrow-direction 3D popup menu.

Surface drawing is also possible without tangible tools, as demonstrated by the *Relief* system from Bourguignon et al. [BCCD04]. In the spirit of *shape from shading* they consider the image plane as a height field to paint on: A closed stroke path defines the ROI for drawing, and the shading controls the (relative) depth: black pushes, white pulls the surface. Absolute depth is propagated from the existing surfaces within the ROI. A problem seems to be the missing continuous feedback: Apparently it is difficult to obtain a smooth surface with discontinuous strokes. An attempt to alleviate this is by a *frisket mode* for masking out depth-irrelevant regions of the existing surface.

Another solution to this problem is proposed by Lawrence and Funkhouser [LF03], namely to paint only the velocity of the depth rather than the depth value itself. This permits to use different colors to specify displacements in different directions: a *propagating velocity* in surface normal direction, an *advective velocity* in a constant direction, and a *curvature dependent velocity* again in normal direction, but in a speed that is proportional to the local surface curvature.

A completely different, or maybe complementary, way to tackle the 3D interface problem is to compensate for the shortcomings of the tangible interface hardware by advanced graphical means. This leads to research in 3D graphical user interfaces (3D-GUI). A classical example is the approach from Grimm et al. [GP95] of a GUI for solid modeling. She points out the inconsistency of adjusting the parameters of 3D operations with a 2D WIMP-style GUI (sliders, curve editors, program code). Much better is to use advanced gizmos, i.e., whole parameterized sub-constructions, that can adapt themselves to the location where the operation they stand for is to be applied. Grimm also notes some basic principles of 3D tool design. This is, most notably, (i) that an object and a tool gizmo should exist in the same space, (ii) that a gizmo provides suggestive handles, and (iii) that the effect of the operation should follow the expectations of the user rather than verbatim the tool parameters: A falsely located/oriented slider in 3D is just as pointless as a 2D slider; intransparent parameters do not automatically become clear just because the controls live in 3D.

Boolean operations and the interactive assembly of complex objects. The choice of the particular representation for solid objects has great impact on the computational cost for CSG. A representation that allowed for interactive CSG already in 1997 was presented by Rappoport and Spitz in [RS97]. Interestingly, their decomposition of polyhedral solids into *convex differences aggregates* (CDA) has also been extensively used in recent computer games. In games terminology a CDA is a set of *brushes*. Each brush consists of a positive cell and, possibly, a number of negative cells. Each negative cell of a brush needs to be contained within the positive cell. A *cell* is just a convex solid polyhedron, i.e., the intersection of a finite number of half-spaces, each defined by an oriented plane in 3D. A brush can simply be rendered using the OpenGL stencil buffer to mask out negative faces contained within the faces of the positive cell.

The CDA is obtained from a special form of CSG tree, one that contains only convex polyhedra. It can be re-arranged in a way that every negative cell is the intersection of two convex polyhedra. Crucial for interactivity is the fact that such an intersection can be computed very efficiently via the convex hull of the dual polyhedron, as demonstrated by Preparata and Shamos: The vertices (a, b, c) of the dual polyhedron are obtained from the brush plane equations in the form $ax + by + cz = 1$. The intersection points are the plane equations of the convex hull of these vertices, transferred back to the primal space via basically the same mapping [PS85]. – A particularly appealing option with interactive CSG is that once the tree is set up, the resulting aggregate shape can be re-computed offline with higher-resolution primitives.

A more fashionable shape representation is the point cloud. Interactive Boolean operations were presented by Adams and Dutré in 2003 [AD03]. They sort the *surfels* (*surface elements*) into an octree for a fast in/out/intersecting classification with respect to the respective other object. Along sharp creases the surface is resampled to improve the crease quality. For the difference and intersection of a helix and a mannequin head, each with about 350 K surfels, they report a rate of 2 fps on a P4 1.6 GHz with a Geforce 4.

A more comprehensive framework of techniques for shape cut-and-paste is *modeling by example* presented by Funkhouser et al. in 2004 [FKS*04]. It comprises (i) interactive segmentation of 3D surfaces, (ii) shape-based search to find 3D models with parts matching a query, and (iii) the composition of parts to form new models. Their technique is easy to learn and able to produce highly detailed geometric models. It permits, e.g., to repair a statue with a missing arm by cutting an arm from another statue, and gluing it to the first, in a predicable, intuitive way. The idea is that the conceptual design for new models comes from the user, while the geometric details are provided by the models in the database.

Procedural shape modeling. Many shapes are generated by distinct, recognizable creation rules, e.g., from the repetition of the same environmental or genetic dispositions. This has implications for the design, for instance, of vegetation: No one would reasonably build a tree by modeling every twig and leaf. Trees are generated instead automatically with grammars and L-systems. Tree growth can be influenced, though. Prusinkiewicz et al. have shown extremely interesting results with the growth of a bush that was constrained to remain within the surface of, e.g., a cow shape [PJM94].

Other shapes that are obviously procedural in nature are rock, berry, mushrooms, and tentacles. Velho et al. have generated them algorithmically with modified subdivision surfaces [VPBY02]. To certain *seed structures* they apply a few times subdivision rules with detail displacements, followed by the original rules to obtain a smooth surface. Crucial is to define/obtain the detail coefficients properly, e.g., by computing offsets from a smooth surface to a fine input mesh by quasi-interpolation. Surface characteristics can also be mixed: With a surface characteristics painting system one might create a berry with a rock-like surface – or vice versa. With lazy evaluation it becomes even infinitely zoomable.

Procedural shape synthesis has recently received much attention because programmable graphics hardware permits to synthesize detail just at the very last stage of output, i.e., during vertex transformation or even rasterization. Great results can apparently be achieved by inflating triangles to triangular prisms that are then volumetrically raytraced by a pixel shader. The surface mesostructure (fine scale surface geometry) can be rendered with *generalized displacement maps* presented by Wang, Tong, Lin, Hu, Guo, and Shum in 2004 [WTL*04]. Their great feature is that they support also non-heightfield mesostructure, such as chain links or rough tree bark, and this even with global illumination. – Lacz and Hart have already presented the first results of their efforts to port great parts of an L-system for generating plants on a GPU, following a parallelized turtle graphics approach [LH04].

1.3.4 Generative Shape Design

The previous sections could only briefly survey the impressive wealth of innovative ideas, of the stunning creativity as well as the sound engineering that have been invested in representing and processing shape, as well as in shape design, during the last twenty years of computer graphics research. So much is solved, but still there appears to be a feeling that 3D graphics has not yet ‘taken off’ the way it deserves. Quite remarkably, the sub-title of the Eurographics 2005 conference is *The evolution of graphics: Where to next?* – Nobody knows, of course, but indications exist that procedural techniques for generating and manipulating shape are among the candidates for a new major mainstream strand in computer graphics.

The attribute ‘procedural’, however, is a bit vague, since this term is used in too many different contexts. An attribute that is a bit less vague is ‘generative’ which, in the context of shape, can be informally defined as follows.

Definition 1.1 (Generative Shape and the Principle of Information Unfolding)

A shape is generative if it has a representation that is not only a list of geometric primitives, but which follows the principle of information unfolding. This means that the shape can be appropriately described by comparably few high-level data, from which a great number of low-level data (e.g., geometric primitives) can be generated, possibly even on demand.

Yet why should procedural, or generative, techniques become increasingly popular and gain importance over time? The reason is that they trade processing time for data size. Instead of unfolding and pre-computing data, data are unfolded from ‘compressed’ data only at runtime. This is possible only when decent computing resources are available – which is guaranteed by Moore’s law to be the case sooner or later. Generative techniques have great advantages since they

- **make complex models manageable** since they allow to identify the high-level shape parameters
- **rely on sufficient hardware resources** since the user will want to tweak the powerful high-level parameters
- **are extremely compact to store transmit** as only the process itself is described, not the processed data
- **allow for model-based reconstruction** which always gives much better results than a blind search, as they
- **make the similarity of modeling and programming explicit** –
which inevitably raises a fundamental issue of describing shape, namely the nasty **code generation problem**

The research on generative techniques is a bit heterogeneous and can not be unambiguously discriminated. The reason is that almost all ideas in computer graphics have certain generative aspects. As an example, consider the primitive *box* expressed in VRML as `Box { size 2.0 2.0 2.0 }`. This is a generative description since it is unfolded to a mesh with 8 vertices, 12 edges, and 6 faces, usually converted again for display, e.g., to 12 triangles. – But of course some contributions stress the generative aspect more than others. A concise description of a box is great, but a pile of boxes has not more structure than a soup of triangles. From this point of view a box is not a generative shape, but only a shape primitive.

Procedural Lego. There are ways, however, to combine boxes that do have more structure than a cube soup. The solution is to follow the ‘psychology’ (the hidden structure) of cubes, which leads right away to – Lego.

In 2000 Anderson et al. have presented tangible Lego-like bricks as a novel input device for 3D modeling [AFM*00]. The bricks have connectors and built-in electronics to communicate their connectivity online to a computer. It can not only display the model currently being built, but it can also *interpret* the model. With the help of an underlying PROLOG database typical shape configurations such as walls and windows can be detected, and even also slanted roofs: One row of bricks is set back by one stud with respect to the row below. Only by detecting and recognizing such structure it becomes possible to *not* render the bricks, but to replace them by a plausibly simulated roof.

Recently Oh and Stürzlinger focused on supporting users building Lego models in a conventional desktop environment [OS04]. As mentioned in 1.1, Lego gains its efficiency through the limitation of DOFs. Brick placement for instance is constrained by the regular rectangular *stud* grid: No suitable fitting studs, no brick placement. This structural information can be exploited for developing smarter strategies for interactive brick placement, e.g., by suggesting only plausible positions when dragging a new brick into (over) the model. Smart selection modes can take into account the direction of a mouse stroke gesture with respect to the orthogonal model CS. An important issue is automatic group selection: A wall should be movable as a whole, not only brick by brick. With intelligent group selection by mouse strokes, to find a plausible new position for a whole group of bricks being moved can be a challenge, e.g., when re-arranging a floor plan.

Interactive Lego and the linear sculpting complexity problem. The problem with interactive Lego is that it does not scale: Every single brick needs to be placed. It shares this problem with all other sculpting approaches. It is only the case that with Lego, the problem becomes very distinct. Smart techniques can drastically reduce the placement time. But for any complicated Lego model the construction time will still be proportional to the number of bricks in the model. This problem can be called the *linear sculpting complexity problem*: The amount of shape information added to a model is only linearly proportional to the number of interaction steps performed by the user. – Michelangelo had the same problem, by the way, since when creating a statue he could remove marble only very slowly, chisel stroke by chisel stroke.

How can the situation be improved? Note that there is one way to create bricks more efficiently, namely the replication of identical parts in a copy-and-multiple-paste fashion. To escape from the linear complexity trap, this facility needs to be generalized to the generation of *slight variations* of a part, instead of only being able to generate identical duplicates. How to realize such shape variations, or variable shapes, is far from obvious, though.

- Placement macros might help to build a whole house, including walls and a roof, from only a ground polygon. Unfortunately, there are many ways to build a house. There are even many ways to build the *same* shape.
- Two-way macros: A brick must retain the information that it was placed by a wall macro. Otherwise the wall can not be edited, since editing involves to remove the previously placed bricks.

Two-way macros directly lead to the fundamental *persistent naming problem*: What if the user chooses to modify one brick of the wall, and then the wall is changed again?

Grammar-based modeling in architecture. Since the 1950's, when Noam Chomsky published his theory on *generative linguistics*, there has been a strong link between the attribute 'generative' and grammars in general. Interestingly, Jeff Heisserman (from Boeing) used grammars for instance in his work on *Generative Geometric Design* from 1993/94 to create the boundary representation of a particular type of domestic houses, the 'Queen Anne houses' [HW93, Hei94]. His approach uses a (somewhat opaque) combination of first-order logic reasoning (`room_adjacent_to_kitchen`), 'solid grammars', Euler operations, and combined operations made from them (`detail_window`).

Besides linguistics and computer science, generative (shape) grammars have a long-standing history in architecture since the early 1970's when Stiny, Mitchell and Gips did their pioneering work [SG72, SM78]. Their ideas are still influential in computer-oriented architecture today, as documented by the yearly conferences of the eCAADe association [eCA], also see the respective summaries from Chase and from Flanagan [Cha03, Fla03].

In the context of computer graphics, grammar-based architectural models are especially valuable for filling urban landscapes in a plausible way, for instance for movie special effects. In 2001, Parish and Müller have demonstrated the automatic generation of the model of a whole city [PM01]. The input to the system is a set of (bit)maps describing the characteristic properties of the different urban areas. These properties (downtown/suburb, population density, etc.) influence the application of parametric rules that further differentiate the city, such as the division of quarters into blocks, the street layout, the building height and style etc. – In their "Instant Architecture" paper from 2003, Wonka, Wimmer et al. have concentrated on detailing the facades of individual buildings, for which they introduced another flavor of shape grammars [WWSR03]. They treat a facade as a rectangular area that is successively partitioned into non-overlapping rectangular sub-areas (floors, window columns, windows), each of which carries certain symbolic attributes. In the very last step these boxes are replaced by previously modeled geometric models read from a database².

One issue with grammars is controllability. To define a grammar is only an indirect way of designing shape. Especially for non-expert users it is difficult to estimate in advance the effect of a certain set of shape replacement rules. Sometimes inconsistent results (doors in the 2nd floor etc) are obtained for reasons that are felt to be obscure by some users. As with all generative techniques it may be that a given rule set behaves falsely only under very specific circumstances, which makes a facility for *grammar debugging* mandatory.

Quasi-generative approaches. The *Procedural Approach to Authoring Solid Models* from Cutler et al. [CDM*02] is often quoted, e.g., by reviewers, as a major reference for generative shape. This may be due to the title. But in fact it proposes a heterogenous input format for a volumetric FEM-simulation, rather than a set of shape generating functions. The VRML-like scripting language permits (a) to load a volume from a file, (b) to wrap volumes with volumetric layers of material, and (c) to define C++ functions 'inline' for custom force or particle functions, which are compiled and linked to the FEM-simulator at runtime. The script language apparently does not support any procedural elements (parameterized shapes, procedures, loops, conditionals), so every material layer requires a few lines of code in the script. As the size of the description is proportional to the number of layers, the description is subject to the linear complexity trap.

Jianer Chen and Ergun Akleman have developed a series of shape generation tools in their research on *Topological Mesh Modeling*, e.g., in [ACS02]. Their focus is on mesh modeling tools rather than on shape description languages. Similar as with the present thesis, they have started from a representation for 2-manifold meshes (based on 'graph rotation systems') with a closed and sufficient set of only two mesh generating functions, `CreateVertex` and `InsertEdge`, plus the inverse operators (their meshes do not support rings, though, see Fig. 4.18). Chen and Akleman have experimented much with patterns of repeated applications of these operators. They found that this leads quite naturally to, e.g., subdivision surfaces and high-genus modeling. Examples of higher-level shape generation tools include (a) the multi-segment curved handle as a pipe-like tunnel between two faces, (b) rind modeling, which can be imagined as turning a wireframe model into a mesh by replacing each wire-edge by a pipe, (c) a variety of exotic subdivision schemes to smoothen a shape, and (d) to smoothly blend two shapes by issuing a single `InsertEdge`, followed by a suitably chosen subdivision scheme.

²Wonka and Müller are currently combining both approaches. Stunning first results on *Transformations in Design* are shown on Siggraph 2005.

Shape description languages: PLaSM and HyperFun. These are probably the principal shape description languages today, each with a vibrant, active user community, and a long record of more than ten years of continuous development. However, neither PLaSM nor HyperFun have become mainstream so far, nor a widely accepted standard. PLaSM was presented by Paoluzzi et al. in 1995 [PPV95] where, remarkably, Snyder’s *GENMOD* from 1992 is cited along with Adobe’s *PostScript* as generative techniques among ‘related work’. – PLaSM is a functional programming language, originally based on the *very high-level* language FL from Backus. It provides a number of libraries with shape operators, many of which can be applied to both 2D and 3D objects. This is because the underlying shape representation uses a decomposition into convex cells which permits using, e.g., Boolean operations in a dimension-independent manner. The shape operations are closed in the space of polyhedral complexes [BBC*99, PPS04].

Despite its expressiveness, its functional power, and many other conceptual advantages of PLaSM, only few references to it can be found in the literature. This may be due to the fact that (a) FL is very different from conventional imperative languages like C/C++, and that (b) PLaSM requires designers to program their shapes. Furthermore, PLaSM code is not easy to generate automatically; so it may be that PLaSM is a classical example for the great impact of the code generation problem. – The PLaSM interpreter is originally written in Scheme, which is another functional language that is also the built-in scripting language of the Emacs text editor, of the AutoCAD 3D CAD program, and the GIMP for bitmap editing. PLaSM is open source, it has no runtime engine but offers an export to VRML (3D models), and SVG and Flash (2D models). A visual front-end to PLaSM has recently become available as well [Pao].

The F-rep/HyperFun project was started by G. and A. Pasko and others under the supervision of T.L. Kunii at Aizu university in Japan in the early 1990’s [PASS95, PA04]. HyperFun is the scripting language for functionally represented shape, i.e., F-rep. Commonly it is known as implicit surfaces, but on the F-rep homepage it is pointed out that strictly speaking, an implicit function $f(x, y, z) = 0$ defines a z -value for a given (x, y) -pair, which is different from the iso-surface $f(x, y, z) = 0$. Due to the generality of iso-surfaces, F-rep can in principle incorporate many sub-representations, from algebraic surfaces to CSG solids and articulated bodies – many of which have indeed been integrated over the years. Great results have been achieved in application areas from volumetric sculpting to cultural heritage, surface reconstruction, and web-based shape modeling with the HyperFun applet.

A conversion of F-reps to meshes is expensive, and interactive inspection and parameter changes are a bit slow, as well as the VRML-conversion of large models; but work on generating optimized triangulations exists [KAP*03]. An alternative is using POVray since functional surfaces are well suited to raytracing. – The HyperFun language, although much closer to mainstream languages, still requires programming for shape design, and the widespread use of F-rep is presumably impeded by the code generation problem, just as PLaSM’s is.

Functional programming and generative animations. *Fran* is a framework for describing virtual worlds as *functional reactive animations* presented by Conal Elliott and Paul Hudak in 1997 [EH97]. Elliott is one of the developers of a famous earlier framework, TBag, that facilitates writing animations in the programming language Clos [ESYAE95]. Clos is similar to C++, but it has a ‘multiple dispatching’ facility that permits to dynamically add member functions to a class at runtime, which is used in TBag to provide objects with behaviours. Another idea from functional languages used in TBag is that values are immutable (‘referential transparency’): When setting $a := expression$, then whenever a is used this is identical to using $expression$ in the same place. Advantages are that there are no side effects and that expressions can be better optimized; the drawback is, of course, an increased memory footprint.

Fran is even more functionally oriented than TBag as it uses the purely functional language *Haskell* developed by Hudak. The design of *Fran* was considered so promising that an earlier version of it was even proposed (by Microsoft) as a candidate to become version 2 of VRML [EI196]. Unfortunately the alternative proposal, *Moving Worlds*, a slightly polished version of VRML 1.0, was accepted as VRML 2.0. – Despite its potential for shape design, *Fran/ActiveVRML* does not provide any advanced modeling operations. It focuses instead on aspects related to interactivity, such as behaviours, events, and it emphasized the continuity of time.

What is interesting about functional languages is that the description of a virtual worlds is usually much shorter than with other types of languages. The reason might simply be that *virtual worlds are functional*, i.e., to use a functional description is just the adequate approach. Note that the present thesis examines a similar idea for generating shape. – Elliott has recently even demonstrated functional programming on the graphics hardware (GPU) with *Vertigo*, a Haskell-embedded language for procedural surface modeling, shading, and texture generation [EI104].

The fact that the graphics hardware nowadays converges towards Turing completeness, to full programmability with loops and conditionals, has most probably a deeper reason. Elliott points out that “*functional programming is a natural fit for computer graphics simply because most of objects of interest are functions*”. As examples he lists parametric surfaces, implicit surfaces, height fields, spatial transformations, vector images (as opposed to raster images), animations, lights, and shaders; the latter can even be understood as higher-level *curried* functions. – *Vertigo*, unlike *Fran/ActiveVRML*, is indeed targeted at procedural shape modeling. Elliott has chosen the composition of parametric curves and surfaces to be the primary device for shape synthesis, which yields very similar results as Snyder’s *GENMOD*.

Shape semantics through metadata and ontologies. Symptomatic for the feeling in the shape community may be the *Aim@Shape* project. It was initiated in 2004 as a *Network of Excellence* (NoE) in framework 6 of the EU-IST programme (*Information Society Technologies*). Its purpose is to stimulate research that brings *shape* and *semantics* closer together, by distinguishing three different levels of shape knowledge [Aim04]:

- *geometric* – such as a scan of human hand, resulting in a point cloud or triangle mesh,
- *structural* – the hand as an abstract hierarchy of bones and fingers, and
- *semantic* – the segmentation of the scan into fingers, associated with bones for animation.

In order to represent a particular interpretation of shape the goal is the “*association of a specific semantics to structured and/or geometric models through annotation of shapes, or shape parts, according to the concepts formalised by the domain ontology.*” (research programme on [Aim04]). The problem the Aim@Shape project is going to solve can probably be stated also in very simple terms as: **What is the result of shape recognition?**

Technologies to be used are primarily *metadata* for shape markup, and *ontologies*, which are a sophisticated device for defining complex classifications, in the simplest case sub-class relations like “a truck is a car”. Also methods exist to define a correspondence between different ontologies, which is useful, e.g., when translating a text from one human language to the other, since all languages tend to have their own hierarchy of notions and meanings. – It will be very interesting to see how the idea of ontologies can be transferred to the domain of three-dimensional shape.

Fundamental reasoning in “A Generative Theory of Shape”. A slightly different view on shape semantics is taken by Michael Leyton in his stimulating and inspiring book from 2001 [Ley01]. Leyton dares to take the very important step from *shape design* to *shape understanding*, following a classical inductive approach: Generalizing a number of significant phenomena he postulates an underlying general theory. Then he attempts to verify the applicability of his theory on examples in a number of fields, as diverse as art and architecture, music, mathematics, physics, psychology, computer science, mechanical design and manufacturing, as well as CAD. As Michael Pratt puts it:

“Leyton is also a cognitive psychologist, whose experiments have convinced him that shape is perceived dynamically rather than statically. In the case of a square, for example, the eye follows one edge, and then intuitively perceives that the subsequent edges are the result of applying the members of a transformation group successively to the initial one.” – [Pra04]

A fundamental hypothesis of his work is the assumption that intelligence always implies to maximize two capabilities:

- *transfer* of actions used in previous situations to new situations, and
- *recoverability* of the sequence of operations that have produced a certain state (or a particular shape)

This means that in Leyton’s eyes *geometry equals memory*. He puts this view in direct opposition to the Erlangen approach from Felix Klein, who defines geometry as the study of invariants under transformation. Leyton objects that this way transformations are ‘memoryless’: they leave the shape unchanged, at least exactly those portions of it that are considered the shape constituents. This impedes the recoverability of operations and, consequently, also the transfer of these operations to other shapes. – Leyton, on the other hand, regards *shape* as some sort of *memory storage for operations*: The square is the result of rotating a straight line, and the line is the result of a pen constantly moving into one direction. Yet of course such a decomposition is not unambiguous, which may be another problem.

Snyder’s pioneering work on Generative Modeling. John Snyder presented the GENMOD approach in a paper on Siggraph 1992 and, in the same year, in his book entitled “*Generative Modeling for Computer Graphics and CAD*”, which is unfortunately out of print today [SK92, Sny92]. GENMOD is an interpreted, C-like language for writing operators to compose parametric functions, seen as general mappings $\mathbb{R}^m \rightarrow \mathbb{R}^n$ of type MAN, which stands for *manifold*. Every operator receives one or more MAN objects as input and produces one MAN as output. An example is the *profile product operator* shown in Fig. 1.10. It combines two manifold objects, two 2D curves, into a two-dimensional surface embedded in 3D, which is also of type MAN:

$$\text{profileProduct} : (\mathbb{R} \rightarrow \mathbb{R}^2) \times (\mathbb{R} \rightarrow \mathbb{R}^2) \longrightarrow (\mathbb{R}^2 \rightarrow \mathbb{R}^3)$$

The parametric surface inherits its two scalar input parameters ‘secretly’ from the two curves; they are not explicitly mentioned. The fact that any MAN object with the correct dimensionality can be used for recombination to produce new MAN objects is the important *closure property* of generative models. Snyder defines a *generative model* as a shape that is created by the continuous transformation of another shape, the *generator*. So GENMOD realizes two levels of mappings: A manifold MAN is a mapping between geometric spaces, and an operator is a mapping between manifolds. But GENMOD is not a functional language, so there are no further meta-levels for creating higher-order mappings or functions. This consideration shows also that higher-order functions would only be a straightforward extension of the generative calculus. Unfortunately this requires a different language paradigm.

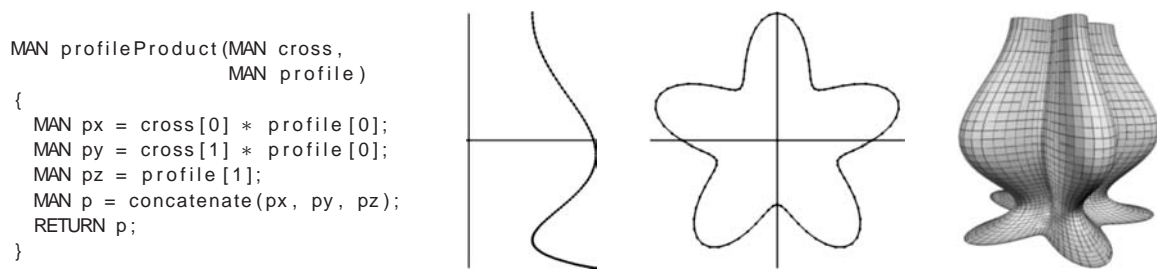


Figure 1.10: Profile product in GENMOD. Parametric surfaces of type MAN are assembled in a C-like language (a). The profile curve (c) is scaled and swept vertically according to the scale curve (b) to produce a surface (d).

The strength of GENMOD, and its most innovative and novel contribution, is that it shifts the attention from surface primitives to reasoning about the assembly of 3D objects, very much in the spirit later proposed by Leyton. The built-in low-level operators can be combined to create custom versions of all the modeling tools that are usually pre-defined in procedural modelers, from sweeping to lofting, rail products, and all sorts of intersections and reparametrizations.

Like all generative approaches GENMOD encourages to write re-usable operators. It is an interesting experience to realize, when writing an operator to model a particular shape, that one is going to solve a certain modeling sub-problem again that is very, very similar to a problem one has solved sometimes before. The immediate reflex is to wonder whether and how the operator for the old solution can be extended to cover also the new problem. This way shape modeling becomes a puzzling problem, as James Kajiya expresses enthusiastically in the foreword to Snyder's book:

“Before this work, I thought about a shape as a collection of polygons, or a sculpted surface. But that view is very limited. *With a sculpted surface there's really no difference between a spoon shape and a chair shape; it's all a matter of positioning the control points in the right places. But a spoon shape has an inner logic, shared by all spoons – and that logic is completely different from that of a chair.*

John and I have spent many hours trying to discover the logic of different everyday shapes. It's an intellectually challenging and exciting endeavour, one that is quite pleasurable when one hits on the right logic of a shape. Because of this, shapes are no longer just inscrutable lumps, but a series of puzzles – sometimes easy and sometimes difficult.”

From the point of view of modeling GENMOD has only two drawbacks: With PLaSM and HyperFun it shares the code generation problem, i.e., modeling requires programming. Second, it provides many tools for creating parametric patches, but it offers no support for stitching several patches together to create manifolds in the mathematical sense.

Under the hood: interval analysis. Every MAN object supports three *recursive operator methods*:

- evaluation $y = f(x), \quad x \in \mathbb{R}^m, y \in \mathbb{R}^n$
- inclusion function $Y = \square f(X) : x \in X \Rightarrow f(x) \in Y$, where X and Y are intervals, and
- differentiation for constrained minimization and root finding

The inclusion function is realized by *interval arithmetic*, which is the second focus of Snyder's book. It is of central importance since it permits to reliably invert a function, i.e., to determine for a given y the set of all points x inside a source interval X that evaluate to $f(x) = y$, basically by recursively splitting the source interval. This way a high-level operator SOLVE can be realized, and together with the differentiation method also another very important operator, MINIMIZE. Both are very flexible, but can also be very expensive to evaluate. They can be employed for constraint solving, ray intersection, implicit functions *and* isosurfaces, for computing the intersection curve between two surfaces as well as to determine self-intersections of one surface, to parameterize a curve by arclength, and many more.

The particular innovation of GENMOD on the technical level is therefore that the program output can be *dynamically queried*. The output is not an image or model, but a set of MAN objects that can be evaluated to produce an image, e.g., by raytracing (solving for a ray-surface intersection), or a model by sampling the surface. The inclusion function permits to bound the region in space where a particular surface will appear. This is a provably unsolvable question with procedural shapes in general (Halteproblem [Tur36]). Only for special cases like iterated function systems it can be solved [LH03].

This GENMOD notion of two-way shape assembly permits for quite advanced methods of shape design, such as for instance exact shape recovery from noisy range data, as demonstrated by Ramamoorthi and Arvo [RA99]. Their method iteratively searches the best fit for the construction parameters of a generative shape template to mimic a given point set. Their system is even capable of automatically choosing the right path in a hierarchy of more and more detailed shape templates. This produces reasonable results even when given the wrong template hierarchy, e.g., for fitting a bowl template to a banan shape, or vice versa.

1.4 Short List of Difficult Problems with Current 3D Technology

The following list is not exhaustive but illustrative. It highlights the most important practical consequences of the shape description problem from section 1.1, both for the practical computer graphics presented in section 1.2 and for the academic computer graphics from the last section 1.3. A potential solution to these problems is going to be sketched in the next section 1.6.

The Semantic Gap between a Shape and its Meaning

Consider a comparison between a scanned page of text and a 3D scanned historic amphora. One A4 page produces 8.7 million pixels when scanned at 300 dpi. These pixels are *not* used, of course, to describe the content or the structure of the document. Instead, the text is extracted using OCR (*optical character recognition*). The scanned pixel image, except illustrations, is usually discarded since it is just an artifact, and it contains no useful information. Remarkably, OCR does not work by matching only individual characters. The recognition rate is greatly improved using semantic information, a dictionary and a catalogue of common syllables – which is nothing but model-based recognition.

Unfortunately, such a canonical method to process the amphora does not exist. Assuming a diameter of 500 and a height of 1000 millimeters, a laser scanner produces more than 1.5 million points on its surface for a millimeter-spaced grid, i.e., to assert a sampling density of (only) 25.4 dpi. A faithful record of all the traces history has left on the surface may be of great interest to researchers – but the most important fact about it is *that it is an amphora*. The extraction of such semantic information from the scanned dataset is possible only if (i) the computer has got a general description of amphorae, and (ii) it is possible to determine whether a given scanned model conforms to it.

The Modeling Bottleneck

All kinds of complicated spatial agglomerations and relationships can be made clear with instructive 3D models. But 3D can be used only when there is something to display. Creating 3D models is very cost intensive. The fundamental problem is the low degree of automatization in shape design. Ironically, the highly engineered domain of 3D modelling still requires enormous amounts of manual intervention to create appealing shapes. The main problems with interactive modeling are:

- **Scalability:** An interactive tool can be applied a hundred times, but not a hundred thousand times.
- **Repeatability:** Slightly different models require re-doing the same work slightly differently again and again.
- **Re-Usability:** The solution to solved modeling problem can not be re-used for solving similar problems.
- **Changeability:** Changing specifications require to further modify the end product of the modeling process.
- **Inter-Operability:** Solutions to modeling problems can not be exchanged between different modeling tools.

Especially the academic approaches to interactive design shape, presented in section 1.3.3, suffer from these problems. For this reason industrial shape design employs interactive tools only when they are carefully embedded into parametric modeling systems, such as the *major CAD packages*, presented in section 1.2.3.

Parametric modeling has also its limits. Note, for example, that the modeling history can sometimes be insufficient: Larger, *structural*, changes require not only to change the parameters of operations, but also the sequence of operations itself. Constraint-based modeling shares scalability problem: No loops or conditionals are available; only the parameters of a feature are subject to constraints, and *not the presence of a feature itself*.

Procedural parametric shape design may solve this, but it implies the *code generation problem* in the user interface: Not all good artists are also good programmers.

Model File Sizes

Shapes become ever more complicated and more detailed. The increase in shape complexity is higher than the increase in hardware capabilities and, more seriously, it sooner or later renders obsolete all 3D methods and software approaches that do not scale. The classical answers to the complexity problems are lossless *mesh compression* and lossy *mesh simplification*, comparable to the difference between the image file formats JPG and PNG [jpe00, png04]. While they may be ideal for scanned models, they have drawbacks for synthetic models, as will be further explained in section 4.1.6:

- Simplification breaks the modeling history: No more changes are possible to simplified shapes at runtime
- Simplification breaks symmetry: Similar parts are simplified differently
- Loss of semantic information: Attached metadata are not preserved throughout the process
- No reduction of shape complexity order: The compressed size is still linear in the size of the input mesh
- Feature-based simplification needs to *speculate* about the nature and importance of shape features.

Digital Libraries of 3D objects

The aforementioned problems become even more drastic when a great number of 3D models is to be managed consistently in a model repository or in a product database. Numerous slight variations of similar parametric models are difficult to manage with traditional database approaches, where 3D models are usually treated as anonymous *binary large objects* (blobs). Emerging standards for attaching meta-information for downstream applications (JT Open, U3D) suffer from proprietary notions a *part*. Exchange of ‘living’ parametric models is not possible with present technology.

Professional systems for *product data management* (PDM) are custom tailored to specific modeling systems, which is why large CAD companies are steadily turning into large PDM companies. They can exploit the inherent semantic and structural information within the respective proprietary CAD format, so that *changeability* and *re-use* are possible within this closed world: A CAD system knows that a door is a door, because the door was created with it.

In a broader context a product database can be seen as a *digital library* containing 3D models. But for the usual primitive-based shape representations (points, triangles, nurbs, etc. as from 1.3.1) it is hard if not impossible to realize the mandatory services required from a digital library: *markup, indexing, and retrieval*. On primitive-based shapes the retrieval problem leads immediately to the general *shape matching problem* which is so hard to solve because it is directly related to the *shape description problem*. Searching for doors or chairs in a triangle soup is very difficult.

Virtual Worlds are Too Static

Shape modeling is a complicated field with its own tools and approaches, and software for interactive 3D is complex also. Furthermore interactive 3D visualization requires decent preprocessing, which breaks the modeling history. Both factors lead to the situation that shape modeling and visualization are decoupled. The fundamental dichotomy between *modeling* and *viewing* may be one of the greatest problems of 3D today. A number of very unfortunate situations arise from the fact that regular viewers do not have modeling capabilities:

- **Paper notes at a design review:** *Power walls* are high-definition projection systems, primarily used in industry, that permit a whole group of designers to discuss details of, e.g., a new building or a new car. Agreed shape changes have to be noted on paper, though. They can not be immediately executed since the link back from the visualization system to the modeler is missing. The inability to try out continuous parameter changes collaboratively leads to insufficient sampling of the design continuum and, thus, possibly to sub-optimal results.
- **No on-line object edits:** Complicated spatial relationships, such as for instance in a *car engine*, can be made clear by an animated three-dimensional engine model with moving parts. An animated model is much better than a static model. For complicated *shape* relationships, as for instance the proportions of buildings, being able to move parts is not enough. Shape animation is what is needed to demonstrate how one shape design evolved from another.
- **Animated characters in computer games:** Deforming things are interesting things. The static surroundings, which can not even be deformed with the biggest guns, are merely the backdrop for the animated creatures in games. How much more interesting games were possible with more deformable shapes, or even editable shapes.

3D can not be used ‘Out of the Box’

It is hardly possible to create 3D content without being an expert user. It is not possible for average people to simply create a shape of an everyday object. – As opposed to creating beautifully styled text documents or to setting up a more complicated spreadsheet.

Shape can not be simply exchanged between normal users in a way that it remains editable across different systems. The situation concerning file formats and data exchange is completely fragmented. Proprietary solutions exist, but lossless conversion between formats is not possible due to a diversity of shape and scene representations. Despite the availability of graphics hardware and substantial processing power the use of 3D is impaired by the lack of suitable software support.

The diversity of tools both for creating and for displaying 3D content is, although favorable in principle, becoming more of an obstacle for the further development of 3D technology: For a novice user there is no obvious way to proceed, and the selection of one technology, which is an investment of time and money, often rules out the use of another. For a software company or an application developer who is not specialized in 3D there is no easy way to integrate a 3D visualization module into an application being developed. As a consequence, 3D computer graphics can not be used ‘out of the box’ – neither by individuals nor by companies.

1.5 Development of the Generative Mesh Modeling Approach

The new method for describing 3D objects proposed in this thesis is the result of a lengthy development process. The chapters to come will only describe the result of this process, the method itself. Some of its properties and features may be slightly irritating at first. But the underlying design decisions were all influenced by the experiences during the development period. It may therefore be worthwhile to briefly outline the whole enterprise, and some of the inevitable detours, in this section. –

Generative modeling is about describing the process of generation itself, and not only its end product.

Initial experience at the KHM/Cologne. The first encounter with the problem of insufficient model semantics and the inflexibility of 3D modeling was at the High School of Arts & Media, the *Kunsthochschule für Medien* (KHM) in Cologne, Germany. A profiled artist, a guest researcher from Holland, who was going to get acquainted with the 3D modeler *Softimage*, had the problem of positioning two spheres in a way so that they touched. She found a practical solution to this problem by positioning the spheres close to each other, zooming in, positioning them even closer, zooming further in, adjusting, etc., until she found the spheres were sufficiently close. As a student of computer science my immediate reaction was to point out that this high-performance workstation was indeed capable of computing the positions of two spheres that are in contact. So although this problem was obviously trivial to solve, to tell the computer to solve it was not. A suitable constraint had apparently not been foreseen by the programmers of the software, and it turned out to be surprisingly difficult to teach the program such wisdom. The problem became really annoying a little later when the number of spheres had to be increased very much: Manual 3D modeling does not scale.

My diploma thesis on generative modeling. On the bookshelf of my tutor D. Schwabe there was one book with a fascinating solution to the problem of expressing the dependency between shapes, *Generative Modeling* from John Snyder. When my advisor Prof. Girod left the KHM I changed back to Bonn to the newly founded computer graphics group of Prof. Fellner. Generative modeling became the the subject of my diploma thesis and the result was a C++ framework to create generative shapes in the spirit of Snyder’s GENMOD with only a few lines of C++ code. Fig. 1.12 (1b,c) shows a tool to generate a surface from input parameters $c_0 \dots c_3$ that are continuous space curves. They are evaluated at the same u -value to produce four points for a Bézier curve that generates a surface point when evaluated for v . The difference to a tensor product surface is that any parametric curve can be used as input $c_0 \dots c_3$. This permits, e.g., to create a normal blend tool (see 1.12 1d,e) that expects two parametric surfaces $s_0(u,v), s_1(u,v)$ and two 2D curves $p_0(t), p_1(t)$. It applies the Bézier blend to the four curves $s_0 \circ p_0, (s_0 + \text{normal}_{s_0}) \circ p_0, (s_1 + \text{normal}_{s_1}) \circ p_1, s_1 \circ p_1$. Based on the same underlying tool a tangential blend can be realized as well (1.12 2d,e). C++ has the advantage of being object oriented. The concept of shape operators maps nicely to a class hierarchy with inheritance of default methods, e.g., for numeric differentiation, that can be over-written with more efficient specialized methods. The disadvantage is that modeling requires not only programming, but even compiling; there is no such thing as a C++ interpreter.

Although C++ is the right language for programming shape operators, it turned out that it is not the right language for programming *with* shape operators. A surface operator, e.g., a bi-rail blend or the profileProduct from section 2 can be seen as a higher-level function. This point of view is not only of academic, but also of practical value. Very much at the end of the diploma period I tried out implementing some basic shape operators in the pure functional language *Haskell*, as an exercise to a course on functional programming I had taken. Very surprisingly, with only 14 KB of Haskell code the expressiveness (but not the more sophisticated numerics) of the much larger C++ solution could be realized. The surface examples in Haskell are shorter than those in C++, as is documented in my diploma thesis (in German) [Hav97].

The project “Verteilte Vermittlung und Verarbeitung Digitaler Dokumente” (V^3D^2). In 1998 the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) followed a proposal from Prof. Fellner to initiate a new research focus concerning the emerging field of *Digital Libraries*, DL. Under the acronym of the German title V^3D^2 it gathered a diversity of more than 20 research projects from several different disciplines for a duration of six years, organized in three phases of two years each. – Besides text documents today’s digital libraries have to manage also all sorts of digital media. These use to come in a heterogeneous and inconsistent bunch of formats, for media ranging from still and video images to CAD construction plans, weather data, and gene databases. The central idea in V^3D^2 was that of a *generalized document* as a unifying view across all types of media.



Figure 1.11: The classical workflow in any public or scientific library.

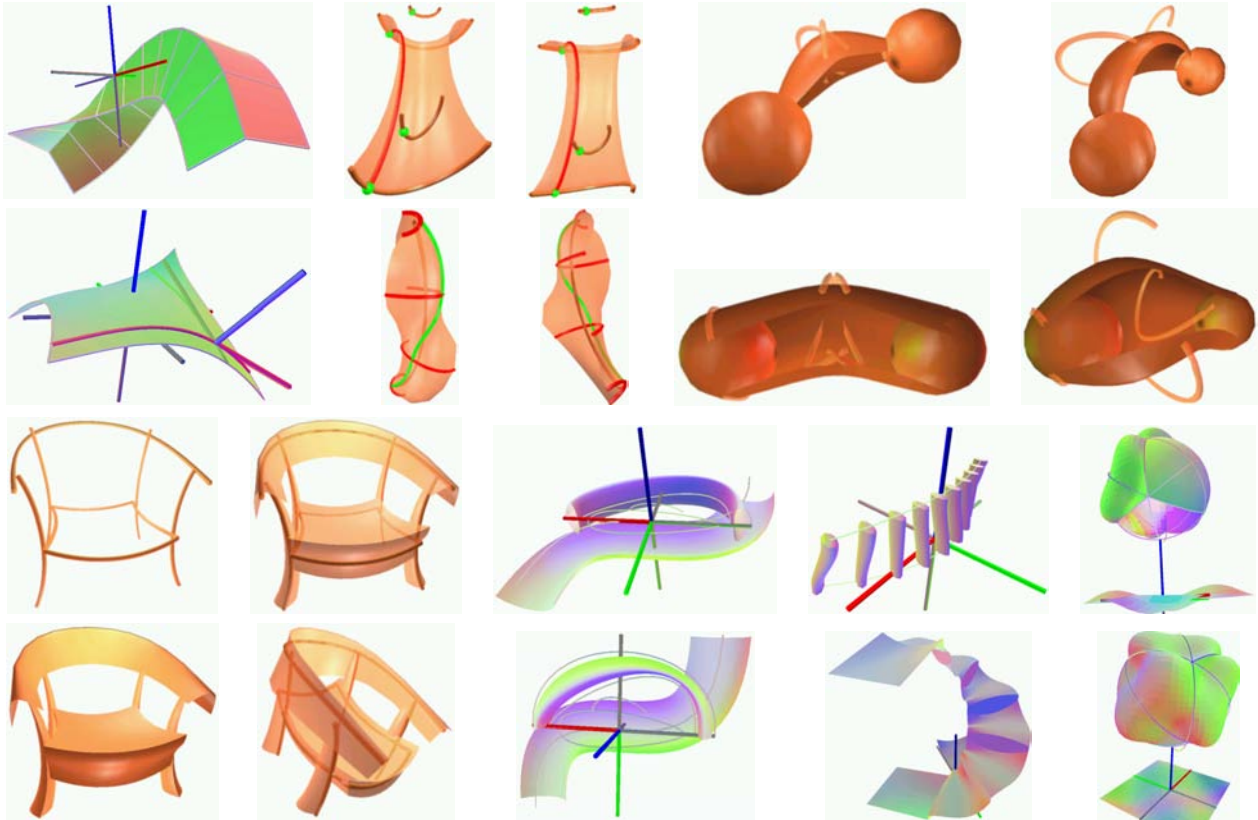


Figure 1.12: Results from the diploma thesis on *Generative Modeling*.

An increasing portion of the data in DLs is three-dimensional. This was the incentive for our group to investigate the question how three-dimensional objects can be integrated into a digital library in a meaningful way. This research was pursued in phases 1+2 of V^3D^2 in the project ModNav3D/ModNav2000 by Gordon Müller and myself.

The classical workflow in a real library is depicted in Fig. 1.11. A digital library follows the same pattern, but instead of human librarians it employs mostly automatic tools along the chain. To integrate a generalized document format, a new media type, basically means to implement methods for *indexing*, *markup*, and *retrieval* of such documents [EF00]. The change of perspective to regarding 3D models as *generalized 3D documents*, which is further explained in [FHH*00], lead to the following hypothesis. We took it as a technical premise for our research:

“A meaningful integration of 3D documents a DL is possible only when as much information as possible is maintained as long as possible in the whole process chain from model generation to visualization.”

For the reasons mentioned in the previous sections this is difficult to obtain with existing model formats: There is no way of realizing a digital library of VRML models without first solving the shape matching problem in a general way. The consequence was to look at alternative ways for describing 3D models.

So while Gordon Müller focused on the automatic construction of spatial hierarchies for efficient navigation and retrieval in 3D [MBH*01, MSF00], I was looking for model representations that support semantic indexing and markup better than primitive-based formats. The natural choice for achieving this goal was to use a generative representation with a paradigm change from *objects* to *operations*. This option is further elaborated and discussed in a journal paper [FHM98] that presents also first results, and in an invited talk at GI99 in Paderborn [Hav99]. The results from ModNav were summarized in 2002 in a workshop paper [FH02].

ModNav3D/2000 and the limitations and prospects of generative modeling. Although generative modeling is appealing in theory it was not at all clear how to realize it in practice, especially with meshes as the targeted surface representation. Existing generative approaches suffer from some severe practical problems:

- limited general applicability, there is no continuous spectrum from primitive-based to generative shape,
- only few surface types supported, e.g., parametric patches [SK92], polyhedra [PPV95], or F-reps [PASS95], and
- the code generation problem: A replacement of interactive shape modeling by programming is hard to justify.

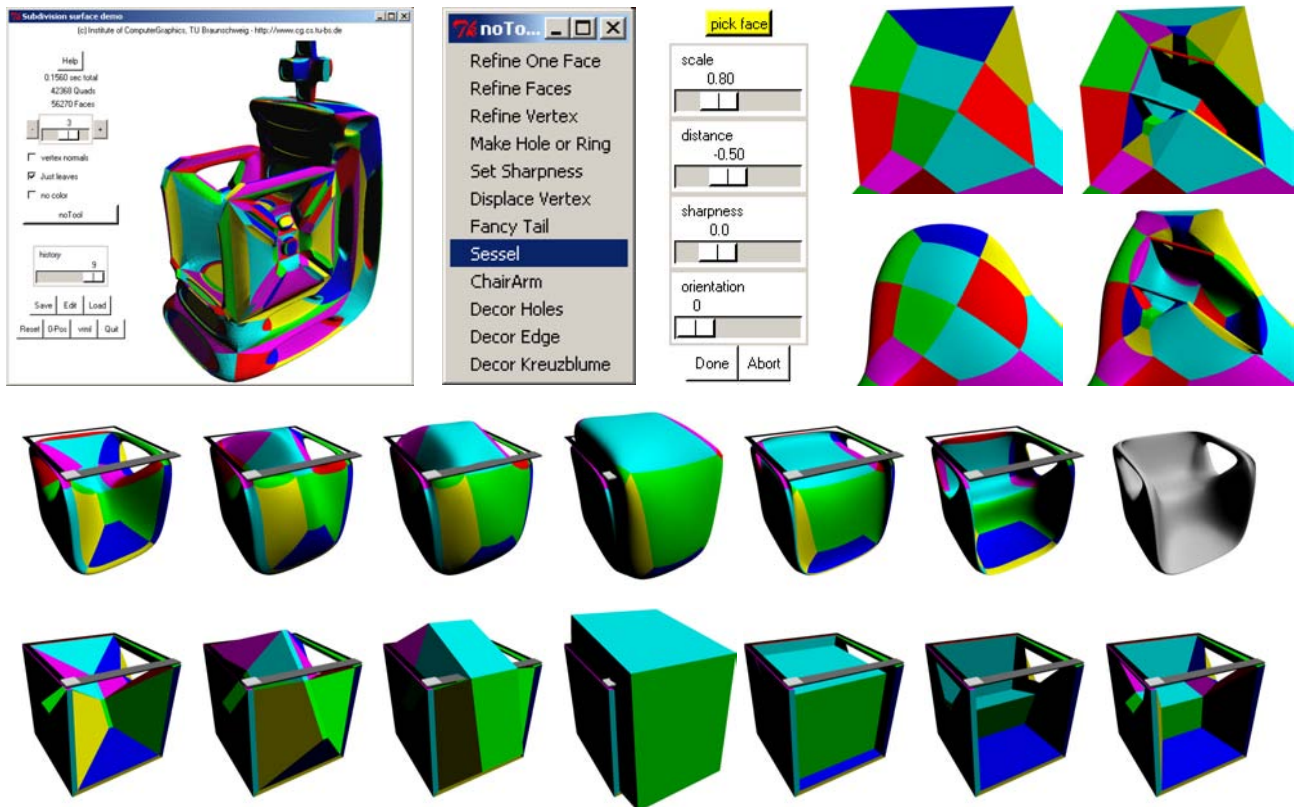


Figure 1.13: The Python-based subdivision surface modeling toolkit. Its *Sessel* ('chair') is not an object but a tool. It is applied to a face, also affecting its neighbours. Specific tool-based DOFs permit for incremental shape design.

These limitations are also challenges, as they are complemented by a great commercial potential of generative modeling. This potential was assessed by the *Gründerwettbewerb Multimedia 2000*, a competition for new business ideas, which has awarded my proposal *4DL – a 3D digital library* with a price of 10 kDM to stimulate the foundation of a new company. The basic idea of this company was to install a three-layer market for trading custom intelligent 3D components. It turned out, though, that for industrial applications compatibility and standards are more important than using the latest cutting-edge technology. The implementation of a new technology in the digital workflow, training employees the fluent usage of the new software, avoiding pitfalls and, most importantly, avoiding mission critical failures, are factors withstanding the introduction of radically new technology – at least as long as this technology is radically new.

From Snyder's GENMOD to Catmull/Clark surfaces: Multi-patch manifolds. The problem of GENMOD is that although it provides a flexible machinery for creating single parametric patches it offers no support for stitching patches together. Practically no interesting objects at all, and especially no real-world objects, can be reasonably represented with one single $[0, 1]$ -parametrization. Not even objects as simple as a sphere can be suitably parameterized as a single patch as, e.g., the parametrization via polar coordinates has singularities at the north and south poles.

This is a well-known fact from topology, and it is the reason why the mathematical definition of surfaces uses the concept of *manifolds*; it will be explained in chapter 2. A 2-manifold surface must provide only *local* parametrizations and not necessarily a single global one. It must be possible to unambiguously and continuously iterate from one local parametrization to the next. The 'overlap' between different parametrizations, or 'charts' of an 'atlas', can be as thin as a common border curve. This leads to a discrete, graph or cell-complex, aspect of mathematical manifolds that is completely ignored by GENMOD. It is often desired that adjacent patches meet not only continuously, but even smoothly. One way to deal with this issue is to develop additional machinery 'on top' of a particular patch representation, e.g., to assert the smoothness of the shape where adjacent NURBS patches meet along a common boundary. This can quickly become very complicated as witnessed, e.g., by the theory of *geometric continuity* (see [HL92, Far02]).

Much more elegant is to use *subdivision surfaces*, e.g., of the Catmull/Clark type. They can be regarded as being a generalization of both polyhedral meshes and uniform B-spline surfaces at the same time; in fact they build a bridge between them. The subdivision surface "virus" was brought to our group with the "*Geri's game*" paper from deRose et al. which appeared on Siggraph 1998 [TDT98].

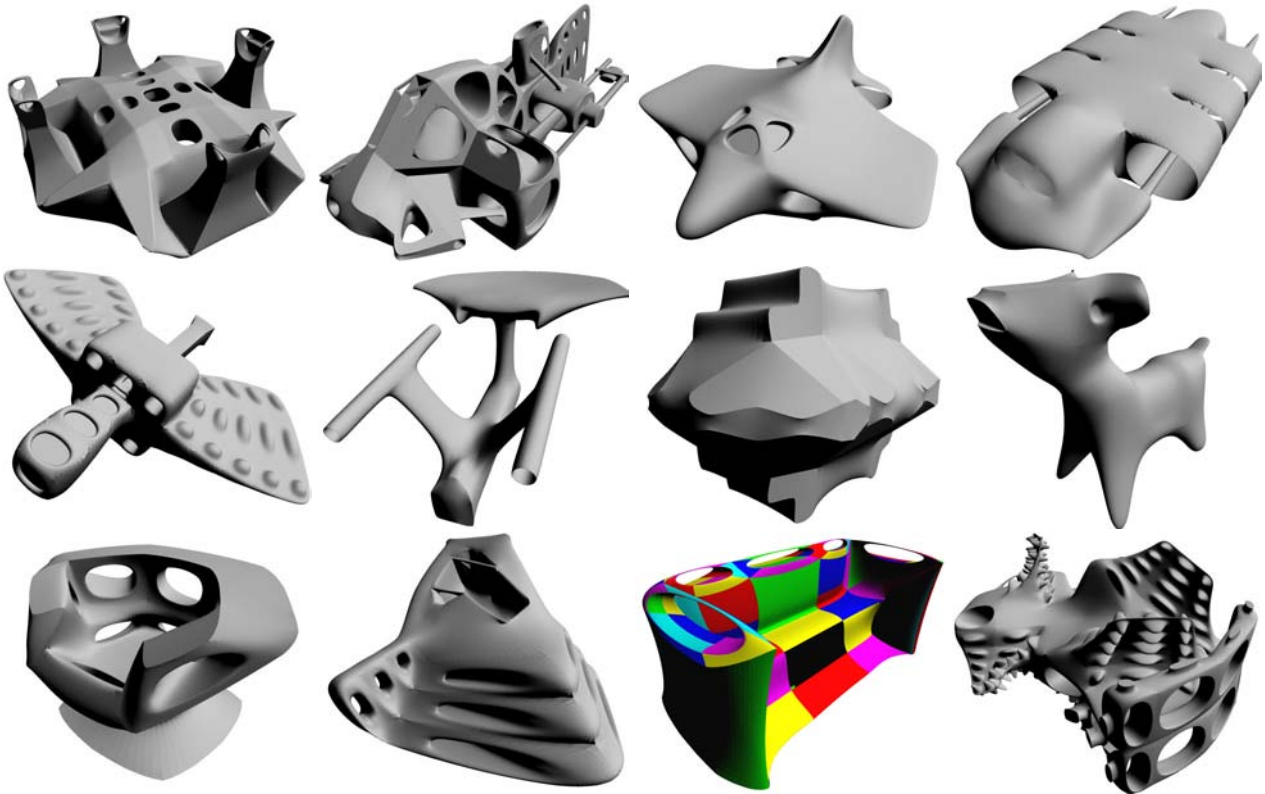


Figure 1.14: Gallery of subdivision surface models. Few high-level tools permit to very quickly obtain models of considerable appeal and complexity. But the set of available tools also limits the range of possible shapes. Not all subdivision surfaces of similar complexity are equally easy to reach in the design space.

Python-based generative modeling with Catmull/Clark surfaces. The first attempt to realize interactive generative modeling was the Python-based subdivision surface modeling toolkit shown in Fig. 1.13. Although this approach had a number of quite innovative features, and was successfully used within some projects in 1998/99, it has unfortunately never been published; instead the experiences gained went directly into developing improved techniques.

The first feature of this approach is that it uses a quadrangle mesh, the *S-mesh* implemented as a C++ data structure, that is then wrapped into a Python extension module. Making use of the *simplified wrapper and interface generator* (SWIG) [B*] most of the member functions of the mesh class could be exported to Python. The resulting direct low-level access to the mesh from Python made it possible to implement very detailed modeling operations in the scripting language. It turned out that this separation is very useful; in fact it is essential:

- The **C++ core** provides the efficiency and it offers only a small interface with a well-defined, immutable set of basic modeling operations.
- The **script language**, on the other hand, provides the flexibility to try out and re-arrange modeling operations with instant feed-back, without any long compilation times.

Python ‘borrows’ also a very flexible *graphical user interface* (GUI) from Tk [O*]. It provides an OpenGL drawing canvas and it permits to easily add and delete GUI elements at runtime. This flexibility is important for re-configuring the GUI according to the currently chosen modeling operation.

Second, an S-mesh consists entirely of quadrangles; no other face degrees are permitted. This helps to keep the data structures very simple. It also allows for a direct, recursive implementation of the modified Catmull/Clark subdivision rules from the *Geri’s game* paper [TDT98]. The special features of these rules are semi-sharp crease edges, by applying the sharp crease rules only a few times and then the smooth rules infinitely often. One floating-point sharpness value is associated with each edge as additional DOF for shape design. S-mesh quad faces hold four pointers to potential children. This tree structure permits for adaptive recursive subdivision to arbitrary depth with a method explained in section 3.3.1. This is ideal for raytracing. A careful extension of S-meshes has been employed with great success for raytracing by Kerstin Müller and Thorsten Techmann as part of their *ShaOLin* (“Shadow Of the Line”) algorithm [MTF03].

The third innovative feature of the Python-based modeling toolkit are the modeling operations themselves.

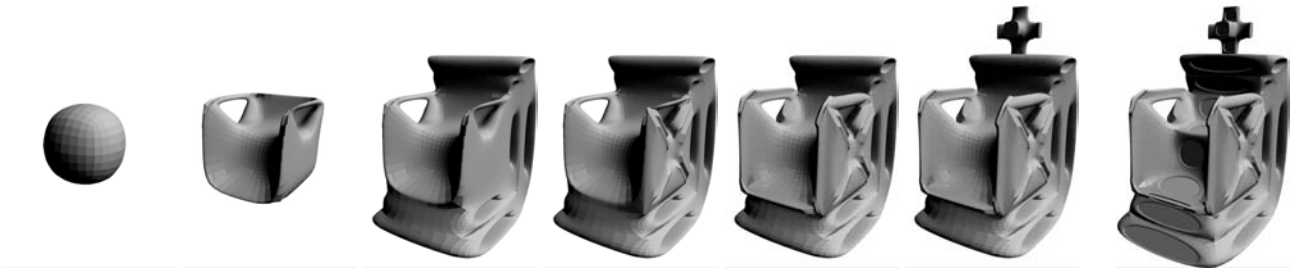


Figure 1.15: The power of high-level tools. The throne can be created from scratch with only 9 modeling steps. Possible is this through (i.) custom tools like the chair and the cross, and (ii.) tools with a builtin loop. They can repetitively crawl over the mesh, like the back/basis in (c) and the side- and middle decoration tools (d,e,g).



Figure 1.16: Simple Lego-style modeling with subdivision surfaces. The cursor with $\pm x/\pm y/\pm z$ arrows can be moved back and forth with pairs of neighbouring keys $q,w/a,s/z,x$. Swift control is surprisingly fast and easy to learn for users with a good spatial imagination. Jittering the control vertices yields a nice rubber-like appearance.

Rapid generative modeling. Shape design in practice with the Python-based modeling toolkit is shown in Fig. 1.13: First a high-level modeling tool *Sessel* ('chair') is selected (b), then it is applied to a certain position in the mesh by a mouse click. This selects a face in the mesh, and within the face one vertex. Clicking on a different position re-positions the application of the tool. This way it becomes immediately obvious to the user what it means that objects are turned into modeling operations in generative modeling. The degrees of freedom of the tool, usually some booleans, integers, and floats, are presented as checkboxes and sliders in a dialogue box embedded within the 2D GUI of the tool (c).

A typical session with interactive parameter adjustment is shown in rows 2 and 3. Preset are the parameters previously used with the same operation; this makes repetitions easier. Users use to start exploring the design space by trying out all kinds of parameter changes. In the case of the chair the design space is the 2-dimensional (*scale, distance*) space, a 2D plane, in which the user traces out an explorative curve. Untrained users get along very well with this kind of interaction although it does not fulfill the three principles from Grimm for 3D GUIs, namely (i) coexistence of object and gizmo in the same space, (ii) suggestive handles, and (iii) design-oriented rather than technically driven parameters ([GP95], see section 1.3.3). Our experience suggests that other ingredients are apparently at least equally important, namely

- immediate feedback of the shape to instantly reflect parameter changes, and
- an undo/redo facility to go back and forth in the sequence of operations already applied.

The naïve play instinct is greatly stimulated when a user changes a slider value of a high-level modeling tool and sees the shape immediately deform and stretch in the most drastic ways. – Without having sound scientific document we concluded that the reason for the positive user stimulus is that the toolkit delivers only few frustration during the interaction. Its shortcomings lie on a different level.

The controllability issue. The gallery of subdivision models in Fig. 1.14 shows the amazing expressiveness of the approach. It is all the more impressive considering the limited number of modeling steps (< 30). The comparably high shape complexity can be achieved within so few modeling steps only with powerful high-level modeling tools.

An engineer from the car industry pointed out, though, that it is apparently very easy to create *some* interesting shapes with this toolkit, but it is not clear whether a designer can realize *precisely* a shape he has in mind. The Python-based toolkit performs only poorly with respect to CAD requirements. This is hinted at by the models in the gallery 1.14 that are supposed to resemble known shapes like the Enterprise (2b), the little cow (2d), the design chair (3a), and the sofa (3c).

```

v0_ray = Ray(Vec3f(4.04,3.99,1.90),
              Vec3f(-34.97,-31.16,-14.53))
v0      = Sessel(0.50,-0.70,0.80,0,v0_ray)
v1_r0   = Ray(Vec3f(-5.86,0.30,1.23),
              Vec3f(48.45,-7.55,-5.54))
v1      = FancyTail(2,1.20,0.45,0,0,v1_r0)
v2_ray  = Ray(Vec3f(3.07,2.19,-3.85),
              Vec3f(-34.17,-19.76,29.81))
v2      = DecorEdge(0.65,0.15,4,v2_ray)
v3_ray  = Ray(Vec3f(-0.14,0.48,2.98),
              Vec3f(7.99,-5.71,-48.21))
v3      = DecorEdge(0.65,0.15,4,v3_ray)
v4_ray  = Ray(Vec3f(1.55,2.03,1.60),
              Vec3f(-21.97,-37.86,-22.59))
v4      = DecorEdge(0.65,0.15,4,v4_ray)
v5_ray  = Ray(Vec3f(1.38,4.37,-0.99),
              Vec3f(-41.62,-36.50,-2.41))
v5      = DecorKreuzblume(0.15,0.35,1,v5_ray)
v6_ray  = Ray(Vec3f(-7.18,0.63,-1.68),
              Vec3f(45.43,3.27,21.25))
v6      = Decor(0.75,-0.03,26,v6_ray)
v7_ray  = Ray(Vec3f(0.47,0.71,4.66),
              Vec3f(-4.76,-7.60,-48.16))
v7      = Decor(0.75,0.10,2,v7_ray)
v8_ray  = Ray(Vec3f(0.64,0.53,-4.67),
              Vec3f(-10.27,-5.37,47.70))
v8      = Decor(0.75,0.10,2,v8_ray)

def apply(self,mesh):
    if self.ray:
        iter=self.ray.apply(mesh)
        for i in range(self.steps): iter.fne()
        i2=lterQS(iter)
        i2.eflip()
        mesh.refineFaces([iter,i2],0.9,0.0,self.sharp)
        mesh.refineFaces([iter,i2],self.scale,
                          self.dist,self.sharp)
        i2=lterQS(iter)
        i2.fne().eflip()
        i3=lterQS(i2)
        i3.fne().fne().eflip().fne().fne().eflip()
        i2=mesh.refineFace(i2,0.6,0.0,self.sharp)
        i3=mesh.refineFace(i3,0.2,0.0,self.sharp)
        i3.eflip().fne()
        i3=mesh.refineFace(i3,0.6,0.0,self.sharp)
        mesh.unifyFaces(i2,i3,1)
        i2=lterQS(iter)
        i2.fpe().eflip()
        i3=lterQS(i2)
        i3.fne().fne().eflip().fne().fne().eflip()
        i2=mesh.refineFace(i2,0.6,0.0,self.sharp)
        i3=mesh.refineFace(i3,0.2,0.0,self.sharp)
        i3.eflip().fne()
        i3=mesh.refineFace(i3,0.6,0.0,self.sharp)
        mesh.unifyFaces(i2,i3,1)

```

Figure 1.17: Python code examples. (a) The *throne* is represented as a sequence of pick actions and operator calls. Code is assembled in background during interactive modeling, and parsed and executed using Python’s `exec` function. (b) The implementation of the *Sessel* (‘chair’) modeling operation as the `apply` method of a tool class.

Swift undo/redo and the ‘magic’ of an interpreted language. The whole modeling history of an object can be scrolled through at interactive speed using a 2D slider (see Fig. 1.13 (a)). The effect of scrolling forward is shown for the example of the throne in Fig. 1.15: It rushes through the evolution of the shape. This is an invaluable device for didactic purposes. It allows users to learn also more complicated modeling tricks very rapidly.

This feature is enabled by the very particular representation of the models created by the Python-based toolkit, namely as *character string with executable Python code*. Python provides a ‘magic’ feature that only an interpreted language can offer, namely to execute at runtime a piece of code stored in a character string. This string is concatenated during modeling. As an example the Python program for the throne is shown in Fig. 1.17 (a): Each application of a modeling tool adds two lines of code to the model. The first line selects a position in the mesh by ray intersection, and the second executes the tool. This execution is completely ‘live’: The Python toolkit can open the model code in a separate editor window. The user can edit the code and the model is updated right away when the ‘apply’-Button is pressed.

Lesson learned: The importance of atomic undo/redo. This experience explains why in our subsequent approaches, and throughout this thesis, so much emphasis is put on *invertible* low-level modeling operations. The Python-based toolkit did *not* have invertible mesh tools. Undo of the last operation is realized by deleting the last two lines from the model string. With every single parameter change, in fact between every two slider ticks, the whole model has to be re-built from scratch (as in row 2 of Fig. 1.13). This puts a tight limit on the admissible number of modeling steps before using the toolkit becomes impractical: With more than around 30 modeling steps the interactivity of parameter changes fades away. Much wiser than building from scratch is of course to un-do the last operation and to re-do it with different parameters. This decouples the undo/redo time of the most recent operation from the length of the modeling sequence.

The creation of high-level modeling tools – the code generation problem. The fundamental problem with the Python-based approach, and the reason why it was eventually abandoned, is shown in Fig. 1.17 (b): High-level tools like the ‘chair’ have to be literally programmed in Python. After having created a dozen modeling tools (Fig. 1.13 (b)) it became clear that this approach does not scale. Myriads of specialized, domain-dependent modeling tools are possible in principle. Really useful tools can be realized on top of the underlying technologies, mesh modeling, subdivision surfaces, and Python. But to assemble the code for these tools like in Fig. 1.13 (b) in a text editor is too difficult and abstract. Lines like `i3.eflip().fne()`, a crawling mesh iterator, are in fact very concrete and also visually imaginable. – So the Python-based modeling toolkit has impressively indicated the importance of the code generation problem.

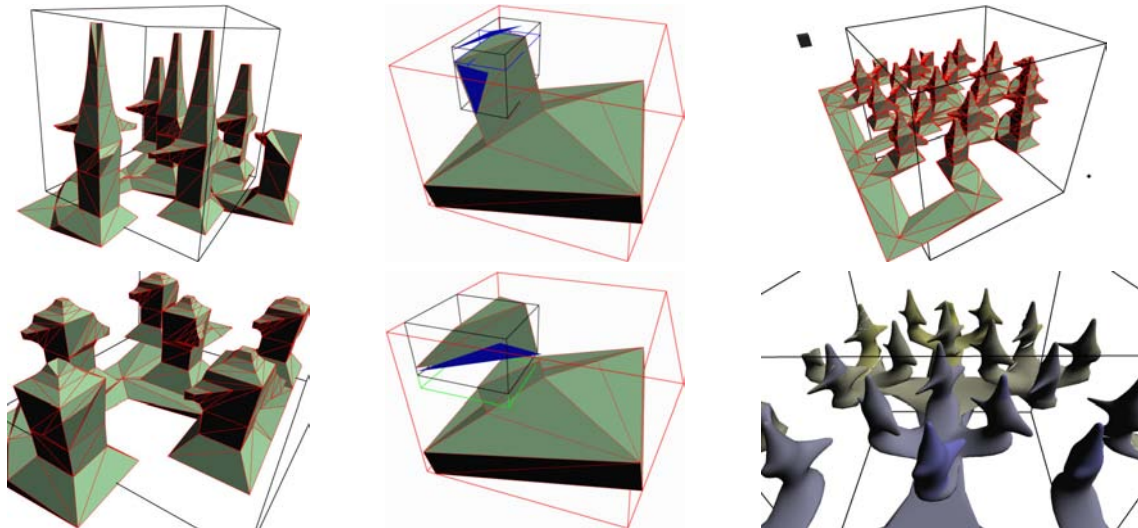


Figure 1.18: Parametric triangle meshes. Triangle splits are recorded during modeling and grouped in macros, shown as boxes. During visualization invisible parts of the model are un-built with collapse operations (1c). The remaining valid triangle mesh can also be shown as adaptive Loop subdivision surface (2c).

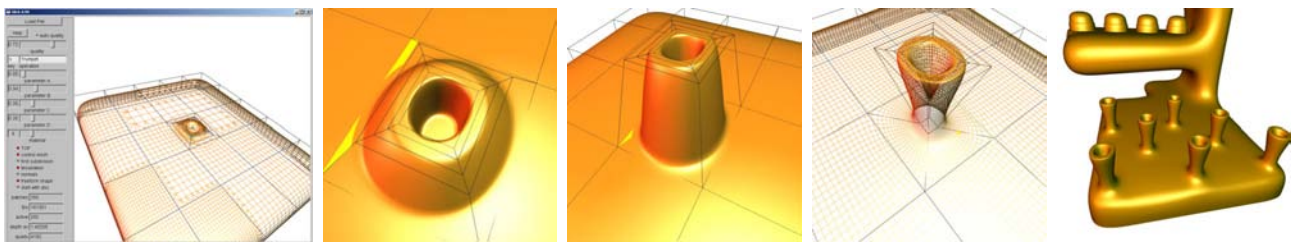


Figure 1.19: Modeling combined B-reps with *Slick*. High-level modeling tools are controlled with (at most) four floating-point parameters in the range $[-1, 1]$, see the sliders in (1a). The adaptive visualization with Combined B-reps and the implementation of mesh manipulation tools in C++ allow for slick high-speed modeling (1d).

Meshes for offline and for online rendering. The experiences with the Python-based modeling toolkit have raised a whole bunch of ideas for improvement. Instead of developing the generative aspect further we chose to concentrate on meshes and adaptive display, in order to create a sound basis for future software layers on top of it.

Data structures for meshes are a delicate issue that is elucidated in chapters 2 and 4. The first template-based half-edge B-rep mesh data structure in our group dates back to 1995 [BFH95]. These *MRT B-reps* from Heinzgerd Bendels were part of the *Minimal Rendering Toolkit* (MRT), an object-oriented raytracing platform for didactic and research purposes. It was originally developed by Dieter Fellner [Fel92] and got considerably extended over the years by a number of advanced contributions such as the radiosity method for global illumination realized by Stefan Schäfer [Sch00, MSF99], or photon tracing for caustics. The modularized, object-oriented design of the MRT even permitted to (ab)use it for antenna field strength prediction by radiowave propagation, as was shown by Norbert Schenk [FS97].

Adaptive display of Loop surfaces. Raytracing and interactive real-time display have slightly different requirements for a surface representation like a mesh. For raytracing a mesh needs to provide a fast ray intersection method. For interactive display, especially with hardware support, it is usually faster to just render a single triangle than to spend CPU time on checking whether it is visible or not (*culling*). The ratio is better, of course, when many triangles can be culled away at the same time with a single operation. Such *patch-based rendering* can provide great savings, especially when simple tests for view-frustum culling and back-patch culling are available.

The article on *Fast Rendering of Subdivision Surfaces* by Kari Pulli and Marc Segal [PS96] was an entry point for Kerstin Müller and myself to the realm of optimized patch-based rendering; the results of our efforts were presented at Eurographics 2000 [MH00]: While Kerstin concentrated on improvements on the 'sliding window method' to tessellate a pair of Loop triangles, my focus was on providing a flexible triangle mesh data structure for it.

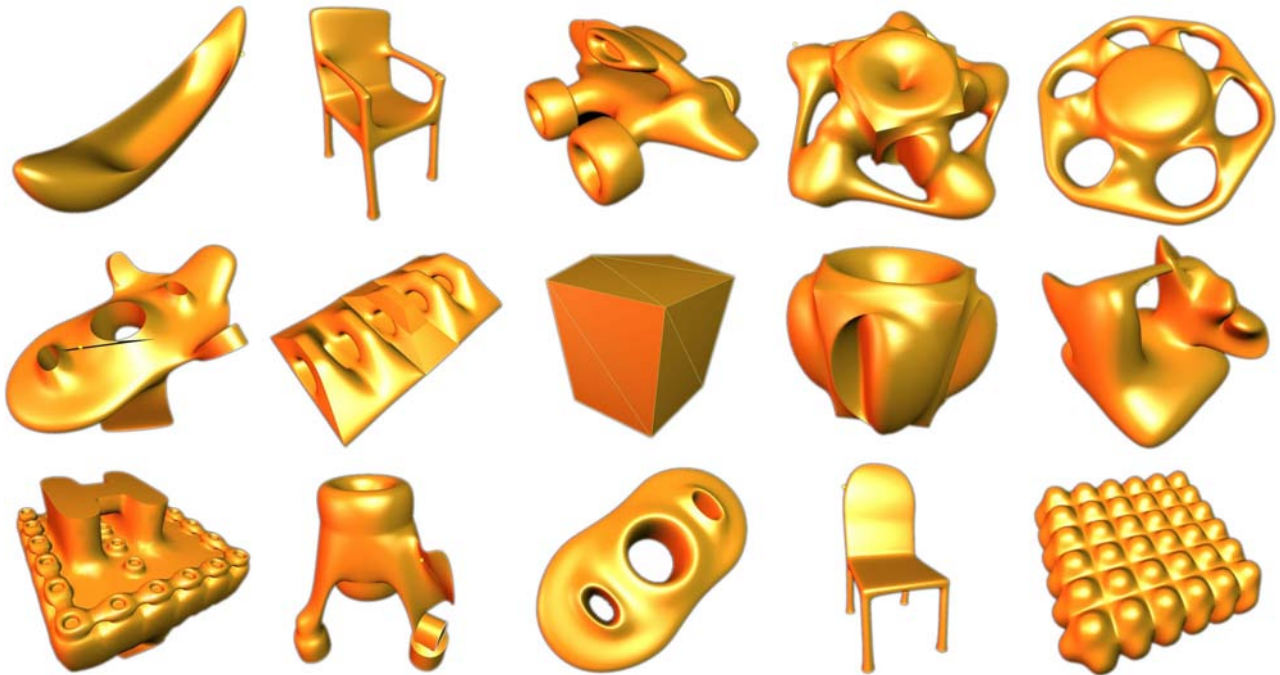


Figure 1.20: Gallery of Slick models. The modeling speed with Slick is very high thanks to the optimized adaptive display with *combined B-reps*. They allow for a combination of polygonal and free-form models (2b,c,d). Only few modeling operations were realized as the main use of *Slick* was as a testbed for increasing the modeling performance of combined B-reps: Interactive manipulation interferes with caching of tessellation data.

Patch-based rendering with template-based mesh data structures. Template parametrization is key because it permits to annotate mesh entities (vertices, edges, faces) with custom data. Very fruitful has proven the view of a polyhedral mesh as a *container data structure*, very much like a doubly-linked list, in the spirit of the *standard template library* (STL) [SL95]. This view is explained in great detail in chapter 4, especially in section 4.1.1. The triangle mesh let Kerstin attach patch data to the triangles. Patch-based rendering of subdivision surfaces means that the control mesh is *not* altered by subdivision. Instead the method does ‘as if’ the mesh was subdivided, but it exploits the regular structure of the subdivided mesh for efficient, strip-optimized rendering. This is much, much faster than altering the mesh by inserting vertices, edges, and faces: Patches are simply stored in 2D arrays where the connectivity is only implicit.

Strategic features. Two further improvements were realized with the triangle mesh approach:

- **Parametric triangle meshes:** All changes in the mesh are logged and can be un-done since an invertible set of triangle mesh modeling operations is used (edge collapse/vertex split etc.). The mesh functions are again exported to Python, a (small) set of high-level modeling operations was realized.
- **Semantic structure with modeling macros:** Macros are a grouping facility for mesh operations to guide undo/redo. Not single modeling operations are subject to undo/redo, but whole sub-sequences, macros. The length of a macro sequence is defined by the user, or by a higher software layer.

The background of these features is the digital library context of the DFG project ModNav2000 (V^3D^2): First, an adjustable undo/redo granularity permits for a *semantic level of detail*: Objects can be simplified in a meaningful way, e.g., respecting shape symmetries. Second, macros have a bounding box, which establishes the link to Gordon Müller’s work on retrieving objects in 3D. Third, macro undo/redo is fast enough to serve as a frame-to-frame culling mechanism, as in Fig. 1.18 (1a) and (1c), where the part of the parametric object outside the query box is un-built. – And finally, all this can also be combined with the Loop subdivision surfaces provided by Kerstin (see 1.18, 2c).

Combined B-reps. Triangle meshes turned out to be much too low-level in abstraction for shape modeling. A quadrangular wall is created with two triangles in a common plane. But there is no way to express this property since both triangles are independent entities. B-rep faces on the other hand can be understood as a facility for grouping individual triangles together. So the new features were transferred back to B-reps.

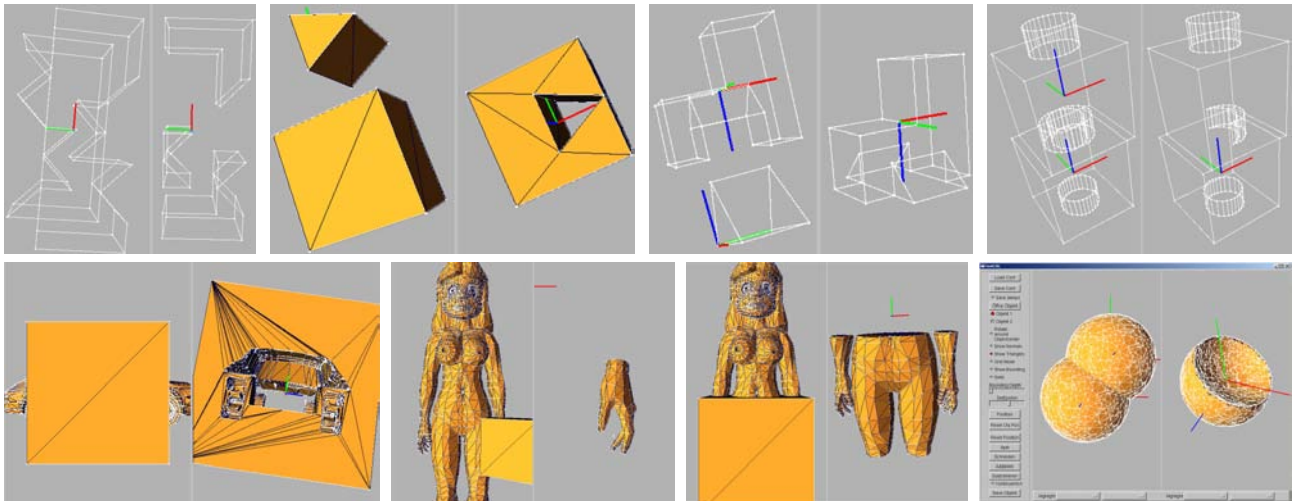


Figure 1.21: *Constructive Solid Geometry* (CSG) on arbitrary meshes. Conventional CSG operates on primitive shapes (box, cone, etc.) stored in the leaves of the tree (1a-d). In his diploma thesis Lars Offen has improved the robustness so much that modeling with cut, copy, and paste of arbitrary meshes becomes possible (2a-d).

Modeling operations for B-rep faces are the *Euler operators* introduced in section 2.2; a corresponding type of free-form surface are Catmull/Clark surfaces. Unfortunately the modified rules we used before for semi-sharp creases from deRose et al. [TDT98] appear to be patented³, so we chose to return to the original Catmull/Clark rules. – Following the same approach as before with the triangles, a B-rep mesh data structure with template parametrization was annotated with patch data (potentially) at every halfedge. Furthermore each halfedge carries a *Boolean sharpness flag*: A mesh with only sharp edges is treated as ordinary polygonal mesh, whereas a mesh where all edges are smooth is rendered smoothly as subdivision surface. The user is completely free to set the sharpness of each edge according to his aesthetical needs.

The resulting data structure is called *combined B-rep*. It is introduced in section 4.3, and a quick intuition of the effect of different edge sharpness settings is conveyed by the *arcade* example in Fig. 4.26. A smooth degree 6-face of a combined B-rep is partitioned into six quadrangular Catmull/Clark patches. Each of these patches has a fixed size of 9×9 vertices and normals for displaying 64 quads in highest resolution. The patch-based approach permits to switch the level of detail at *no cost* at all on a frame-per-frame basis, by switching between certain pre-computed OpenGL triangle strips. The second vital feature of combined B-reps, and the reason for their name, is that they bridge the gap between polygonal and freeform modeling. In fact they combine both complementary ways of modeling within the same data structure. Combined B-reps were introduced in a journal paper [Hav02b].

Progressive combined B-reps. The third novel feature of combined B-reps is that they allow for *selective tessellation updates*. This is the solution to a very nasty technical problem: Interactive shape manipulation interferes with caching of tessellation data. The user may change the mesh anywhere, add or delete entities, even switch between polygonal or freeform. The tessellation of the surface must be quickly re-computed in all affected regions – and, which is more important, *only* there. This feature is the basis for efficient undo/redo when changing, e.g., one window in a complicated facade. As with the triangle meshes, undo/redo was realized with macros of elementary operations, which yields in this case *Euler macros*: Euler operations are complete and sufficient on the class of B-reps, and they are also invertible.

A slight complication arises from the fact that, unlike with progressive triangle meshes, also *destructive* operations can be used within an Euler macro; this requires special care with undo/redo. This much more sophisticated version of a combined B-rep with logging of Euler operations, Euler macros, and semantic LOD, is therefore called *progressive combined B-rep*. It has been submitted for publication [HF05a].

Alternative ways of modeling: CSG. In parallel to the combined B-reps a number of other modeling approaches have been pursued. The diploma thesis of Lars Offen [Off03] was about an improved CSG implementation based on the method proposed by Martti Mäntylä in his book [Män88]. The ambitious goal was to realize CSG on arbitrary B-rep meshes with only 32 bit single-precision floats for vertex coordinates. This is difficult because first, no exact knowledge on the type of input objects is available, in contrast to the usual CSG operating on primitives (box, cylinder, etc). Second, B-rep meshes are more general than triangle meshes as they can have higher-degree faces and also faces may have rings.

³oral communication with Denis Zorin

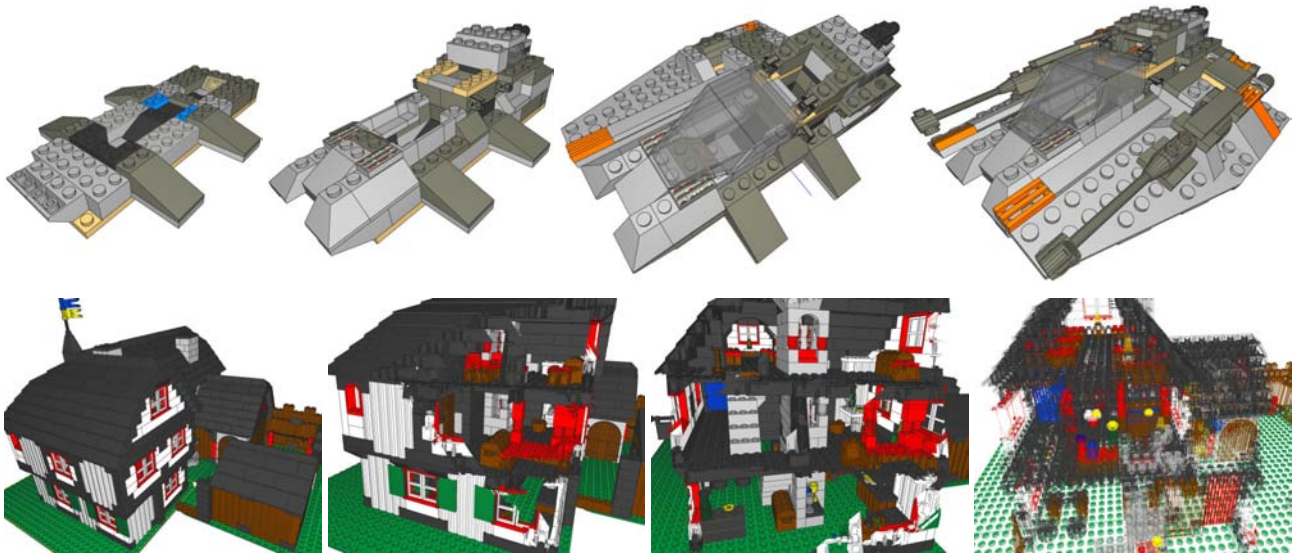


Figure 1.22: Lego models from the LDraw community. The .dat-files can be used as interactive Lego construction manuals as they permit to store *stepping information* within the model. The highly performant Lego viewer created by Felix Funke (as his Studienarbeit) has a stepping slider. Problematic is the invisible geometry (2d).

Despite the considerable success shown in Fig. 1.21 it turned out that accumulating numerical inconsistencies can be a very serious problem. This has increased our sensitivity to numerical problems. It also highlighted the subtle fact that in a floating-point universe it is not so easy to determine whether a given point is above or below a certain plane. The fundamental problem is that higher-degree faces are practically never planar; the fourth point of a quad is generally in a non-zero distance to the plane through the first three points. Working with tolerances is unavoidable, for instance faces need to have a thickness. In order not to let a single ‘thick’ face spoil all the rest, and make for instance very small faces vanish, every face even needs to have its own thickness. Moreover, this has implications for the ‘thickness’ of edges and vertices. – In response to this problem Lars has developed a number of sophisticated mesh repair tools.

A second result was that the CSG method can be decomposed into single modules that are useful in their own right: An intersection routine that expects a pair of intersecting faces (from one or two solids) and follows the intersection to determine a closed *cut path*; a cut routine that dissects the surface along a given cut path; an inversion routine that exchanges inside and outside; and a method for gluing two surfaces together along a given path. A subtlety of CSG are co-planar faces; Markus Lagemann has shown in his Studienarbeit how to perform 2D-CSG with a sweepline algorithm [Lag04] to solve it. Both this and Lars’s work are exclusively using Euler operators. No direct manipulation of mesh entities was needed, which shows the suitability of this interface.

Alternative ways of modeling: Lego. The world of *system shapes* opens the fascinating field of *meta-shape-design*. The general idea is to design a limited number of basic elements from which an unlimited (or at least very large) number of shapes can be built. A multitude of examples for system shapes exist that are interesting in both a theoretical and a practical sense. Theoretically interesting is how to design a shape system as a ‘meta-shape’ – practically the question is how to realize a given shape with a given system.

The prototype of a system shape is, of course, Lego. The LDraw community has faithfully collected an archive of all known Lego parts on ldraw.org, and re-created many classical Lego models with them. The LDraw archive was also used in the *Studienarbeiten* of Tomas Neumann and Felix Funke. Their task was to create an interactive Lego viewer. Impressive results of this exercise in optimizing OpenGL display are shown in Fig. 1.22. Lego models are visually rich and very suggestive and at the same time comparably lean, as they consist only of a list of parts, each with an ID, orientation, and position. The problem is only that huge amounts of invisible hidden geometry are created this way (see the wireframe rendering in 1.22, 2d). To remove them is not trivial.

The drawback of such a ‘naive’ realization of system shapes was already mentioned, namely the *linear information complexity trap*: In terms of complexity classes there is no difference between a list of Lego pieces and a corresponding list of triangles. – The link to the shape description problem, and the solution of the complexity issue, is the following observation: *In almost all Lego models the position of almost all pieces is not random.* Furthermore, when system shapes can be understood as meta-shapes, is it also possible to devise a *system* for creating *system shapes*, i.e., a meta-meta-shape? And if so, could the Lego idea be useful for it – as some sort of *meta-Lego*?

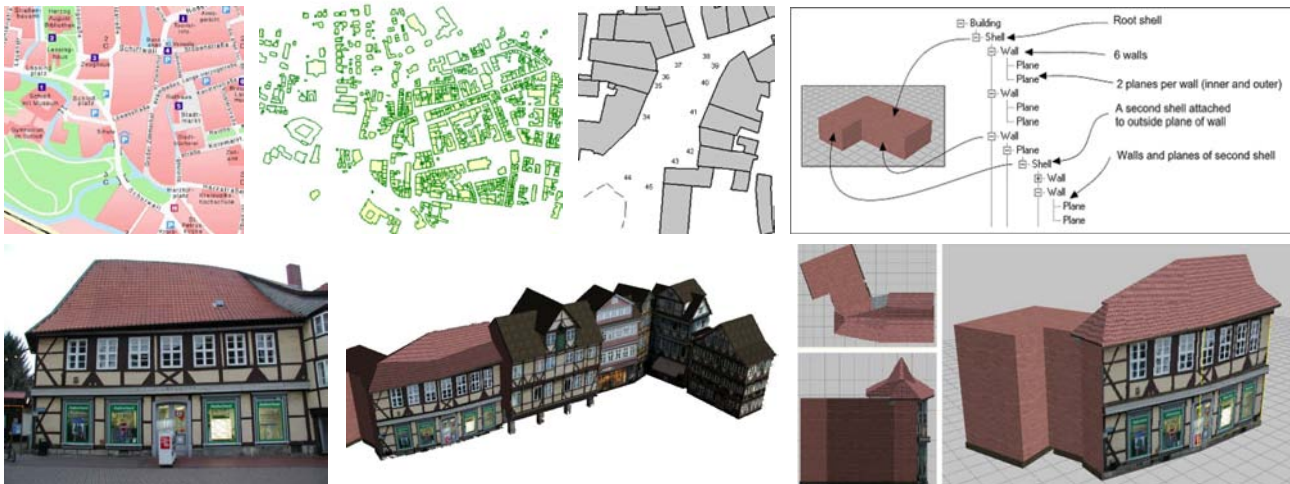


Figure 1.23: The Charismatic workflow for the Wolfenbüttel reconstruction. Conventional plans and official landmark data are combined with photographs taken from strategic viewpoints (1a-c). The ingenious Charismatic shell approach (1d) permits to quickly recreate agglomerations of faithfully textured buildings (2a-d).

The Charismatic project. The vision of *Charismatic* was to re-animate deteriorated cultural heritage sites. The name is an acronym that stands for the impressive title *Cultural Heritage Attractions featuring Real-time, Interactive Scenes and Multi-functional Avatars as Theatrical, Intelligent Characters* – and this title is indeed the programme. Charismatic was funded by the EU in framework 5 of the Information Societies Technology (IST) programme in 2000-2002 with a consortium of eight groups from UK, Greece, and Germany. The technical core partners were Prof. David Arnold's group at the University of East Anglia (UEA), Tim Child's television company *Televirtual*, our group, and the speech recognition and text understanding group from British Telecom (BT) contributing software to support intelligent yet simple conversations with a virtual tourism guide.

The vision of Charismatic was to install a new VR tourism industry to exploit Europe's unique position in global cultural heritage. The key idea was to produce tools that permit an *economically viable* production of complete virtual reconstructions of populated sites of historic and cultural interest. Impressive examples of virtual reconstructions exist already; the problem is only that practically all of them have been created with substantial funding. Funding of pilot projects is great to prove that a novel idea can be realized in principle.

A massive breakthrough is only possible with the economic power of a commercially interesting market for cultural heritage content, when the new area becomes an industry. Exchange is possible only with standards. – Great economic potential exists in the form of location-based entertainment in visitor centers, where ancient worlds are re-vitalised in the most impressive and engaging ways. The focus of Charismatic was therefore not only the modeling and interactive rendering of urban environments, but equally tools for intelligent avatars and interactive storytelling. The latter areas were worked on by Televirtual and BT, while UEA and our group focused on the modeling and rendering of urban environments.

The UEA scene graph versus OpenSG. All existing rendering engines have shortcomings. The UEA group enthusiastically started to implement their own renderer, the *UEA scene graph*, with all kinds of optimizations for urban sceneries:

- ROAM engine for adaptive terrain rendering with LOD control
- The 'occluder shadow' method to cull away buildings hidden behind other buildings
- Impostors to replace buildings at a distance by a single textured polygon
- Buildings with attached detail geometry that can 'retract' into the building when seen from a distance

The use of a proprietary scene graph turned out to be a major problem at the end of the Charismatic project: Exchange requires standards. Fortunately our group was involved in an open source scene graph initiative, the *OpenSG plus* project to extend the open source scene graph *OpenSG* [Opeb, Opec, RVB02]. But unfortunately, OpenSG was available only in a very first, unstable version by the time when Charismatic started.

The highly optimized routines from the UEA scene graph have nevertheless been preserved, as they have finally migrated to OpenSG as part of the *Epoch Network of Excellence* in 2004. Epoch also promotes OpenSG as common rendering platform among a consortium of more than 90 groups to assure that all 3D software modules are inter-operable.



Figure 1.24: Charismatic: Comparison between real and virtual Wolfenbüttel. The liveliness of historic VR was greatly improved with detailed surroundings (1c) and theatrical elements, avatars and interactive storytelling (2c).



Figure 1.25: Charismatic: Wolfenbüttel houses. Shells can be greatly distorted, a feature that has proven very valuable with historic buildings. With each house also one special feature was realized, like an arkade or a balcony.

The Charismatic production pipeline and hybrid model representation. The workflow from the available primary data to the reconstruction of Wolfenbüttel is depicted in Fig. 1.23. To measure success we chose to first reconstruct an existing city with three students in our group using the UEA shell modeler, results are shown in Figs. 1.25 and 1.24. The following model representations were used within Charismatic:

- **UEA shell models** for domestic houses reconstructed using the *Charismatic Shell Modeler*, see Fig. 1.23 (1d,2d)
- **UEA mesh objects** for landmark models (churches, fountains) created externally with, e.g., 3D Studio Max
- **TU-BS combined B-reps** to represent ornamental free-form detail with built-in level-of-detail (cf. Fig. 4.30)
- **TU-BS progressive meshes** for adaptive LOD rendering of dense triangle meshes from scanned artifacts

The usefulness of combined B-reps for architectural detail was demonstrated at VAST 2001 [HF01]. The architecture of the Charismatic software was presented in detail in 2003 at the VAST and Cyberworlds conferences [HFDA03, DAHF04].

The key to success: Semantic information. Charismatic gave us the opportunity to transfer the theoretical approaches from ModNav3D/2000 to an applied practical setting. Both modeling and rendering a complete city are hopeless without an approach that *scales*; ideal conditions for generative modeling, which convinced also our project partners.

Key to success was the strict adherence to the working premise of ModNav3D/2000, the maintenance of structural information as long as possible. Charismatic has shown that **both modeling and rendering** greatly benefit from a high-level shape representation. A shell can be compactly described with only a few floats. Similarly, a complete virtual reconstruction can be described with few but well chosen and structured data. Few DOFs mean that modeling goes fast. It also helps the renderer because it knows much about the objects it is going to display. The renderer can make efficient use of the data to choose the appropriate LOD at runtime and generate derived data (impostors etc.) on the fly.

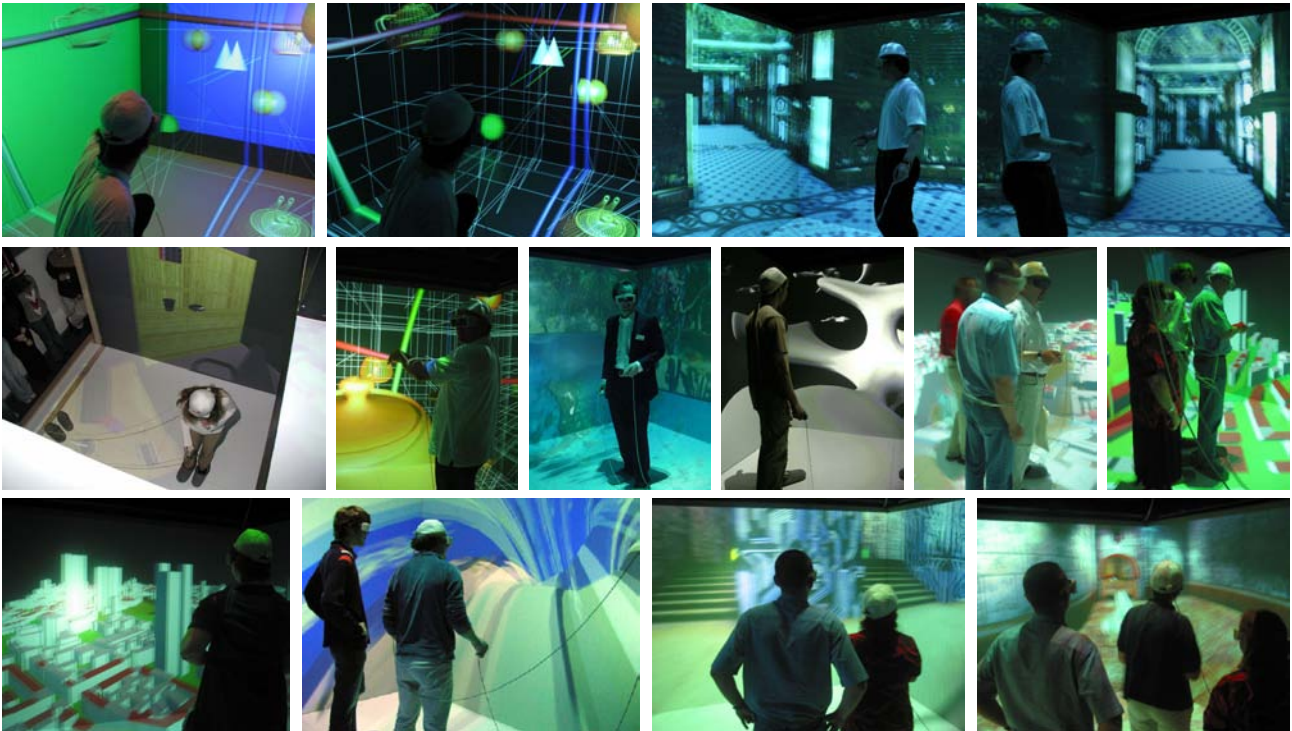


Figure 1.26: Viewer applications in the DAVE. The ‘teapot’ demo (1a-b,2b) is part of the Davelib to demonstrate the easy programming of the DAVE. The fishtank and the Rhino-NURBS-viewer (2c-d) are interactive DAVE applications. Our OpenSG-based VRML viewer (1c-d,2a,2e-f,3a-b) is very impressive with architectural models. It is regularly used for design reviews by architects (3b). Very high quality is possible with specialized walkthrough-software such as Cave Quake III Arena (cq3a) that was ported to the DAVE using the Davelib within a day (3c-d).

The DAVE. The ultimate high-end in virtual reality is marked by so-called *Cave*-systems. They let the user *immerse*, i.e., dive into, a visual computer-generated reality that completely surrounds him. It allows to freely move within a space of typically 3×3 meters and to look into any direction. The projected view is immediately updated accordingly. The stunning impression of *quasi-tangible 3D* is caused by the fact that the body motion is synchronized with the view: We all have many of experience with body motion, and we have precise expectations on how our view changes with every movement of our body. The understanding of spatial relations in a Cave is greatly enhanced by the sense of the own body.

The particular Cave system we realized in Braunschweig in 2002 is called DAVE, which stands for *Definitely Affordable Virtual Environment* – compared to other systems it is inexpensive but provides better quality than Caves with CRT projectors. Our goal was to build a four-sided Cave on our own from off-the-shelf standard components [Hav02a, FHH03].

A spectrum of immersive techniques. The immersive effect is achieved by a stereoscopic surround-projection combined with head tracking. This means that the computer knows at any given moment the position and orientation of the head of the person in the Cave. Such a system is at the high end of a whole range of possible projection systems:

- **active stereo on a desktop computer:** Images for the left and right eyes are time-multiplexed RLRLRL... and a pair of *shutter glasses* blacken one and then the other eye’s view synchronously. This is called active stereo because the glasses need to actively switch between states. The sync signal is usually transmitted via infrared light.
- **single-screen passive stereo projection** is possible with a dual-headed graphics board (for two monitors) feeding two projectors that project their images to the same screen. A pair of polarization filters in front of each projector and a corresponding pair of filters in the passive glasses make that the eyes receive images from different projectors.
- **single-screen passive stereo with head tracking:** A perfect stereo view is maintained for one person. The user can also move in front of the screen and stick his head into, e.g., the 3D model of a car engine.
- **multi-screen active stereo with head tracking (Cave):** Several projection screens can cover a wide field of view, ideally the *whole*, as in a 6-sided Cave. This permits to inspect and walk *around* objects freely floating in space.

Only a single person has a perfect view in a Cave; persons standing close can usually follow easily. Like for the stereo, several options exist also for the tracking. *Optical tracking* uses reflective markers, infrared light, cameras, and image

processing to determine position and orientation. *Electro-magnetic tracking*, which we use in the DAVE, works with a *sensor* that measures a pulsating magnetic field created by an *emitter*. It is less expensive than optical tracking and robust against occlusion. But the measurement can be disturbed by any piece of metal that is closer to the emitter than the sensor. Unfortunately it turned out that the room for our DAVE was heavily armoured with steel, especially around the hole in the ceiling which we needed for the image on the floor which was projected from above.

The DAVE hardware: Active stereo with digital DLP projectors. Passive stereo, e.g., linear polarization, is not rotationally invariant: Tilting the head by 90 degrees reverses R/L. This is a problem for a Cave, especially on the ground. Active stereo does not have these problems, but unfortunately it is normally *not possible* with modern digital projectors. The reason is that, e.g., DLP projectors create time-coded colors using a colorwheel and a million binary mirrors, one per pixel. This raises delicate timing issues. So the projector runs at a fixed 120 Hz, irrespective of the input frequency: The projector has its own framebuffer. So feeding a digital projector with active RLRL signals produces just garbage.

The great technical innovation of the DAVE is that it realizes active stereo with off-the-shelf DLP projectors. How is that possible? This solution is found, realized, and patented, by Dr. Armin Hopp, an alumni of our group, now head of digital image. The key is to modify the projector electronics, the *formatter board*, so that the projector becomes *externally synchronizable* (ethernet cables in Figs. 1.27, 4a-d). Armin manages to switch off the framebuffer of each projector for the duration of every other frame, alternatingly for a R/L pair, which yields comfortable 60 Hz per eye. The CPUs of all eight projectors are kept in exact sync by a central device, the synchronizer (1.27, 4c). It also steers the six arrays of infrared LED from Torsten Techmann that drive the shutter glasses (5d,e) by flooding the DAVE with flashes of infrared light.

Sustainability by two levels of synchronization. The schematic overview in Fig. 1.27 (1a-c) shows the projector synchronization in yellow and the standard ethernet to synchronize the eight DAVE clients in red. These clients run in parallel eight instances of the same (deterministic) program. The only difference between the program instances is their OpenGL projection matrix, which is set according to *screen* (left,front,right,ground) and *eye* (R/L). The computers are synchronized only at the level of *buffer swaps* (at 30-50 Hz) issued by the daveserver via UDP broadcast. The server runs the ninth program instance, and it is connected to the tracking system. With every broadcast it sends the new position/orientation from the 6DOF sensors of head and mouse to the clients.

This is the key to sustainability for the DAVE: The image *display* is synchronized instead of the image *generation*. Active stereo usually implies using dim and expensive projectors based on the old *cathode ray tube* (CRT) technology, where the VGA signal from the graphics board directly steers the electron beam in the tube. To drive a CAVE then requires very special graphics boards with a *genlock* that permits to exactly synchronize the VGA signals from all boards. But with our DAVE we can always use the latest generation of off-the shelf high-performance graphics boards. This means that we can upgrade all eight of them every year, and the eight PCs every two or three years, and still remain within our budget.

The DAVE software: The Davelib to port legacy code, and OpenSG. The Davelib is a C++ library to facilitate the quick adaptation of any given OpenGL program to the DAVE. In the simplest case it is sufficient to allocate an object `dave` of the class `Dave` and to insert two lines of code into the `redraw` routine (which every OpenGL program has), namely `dave→setProjectionGL()` before and `dave→waitForSync()` after rendering the scene, and to recompile the application. Since its initial version, which I wrote in 2002, the Davelib encapsulates all the essential software components to drive the DAVE in the singleton C++ class `Dave`:

- on the server: reading the head/mouse tracking data from the Ascension *Flock of Birds* via serial bus (RS 232)
- on the server: correction of the tracking data via `libTrCalibr2` (from the university of Utah),
- communication and synchronization between server and clients via UDP broadcast
- on the clients: setting the OpenGL projection matrix for the current view, and access to the hand and mouse CSs

When adapting a desktop OpenGL program to the DAVE, much more involved than the rendering is usually to replace the 2D mouse by the pair of 6DOF input devices used in the DAVE. The 2D mouse may be used for many purposes, from selection and picking over parameter adjustment to 3D navigation. For a deep integration all this functionality must be realized by the combination of head and mouse position and orientation, which are 12 DOFs and not only 2. A very acceptable default solution is to emulate a 2D mouse pointer. The mouse arrow is drawn on the screen at the point of intersection with the ray from the tracked head into the direction of the tracked 6DOF mouse.

An alternative to direct OpenGL coding is to develop 3D applications on the *scenegraph* level. The *OpenSG* system is an open source scenegraph with a unique feature: It supports interactive rendering of a dynamic scene jointly on a *cluster* of computers. Local copies of common data on are kept consistent across different machines, changes are committed at common synchronization points, typically once per frame. Due to its compatible design OpenSG could be adapted to the DAVE within a day by Christoph Fünzig using the Davelib.

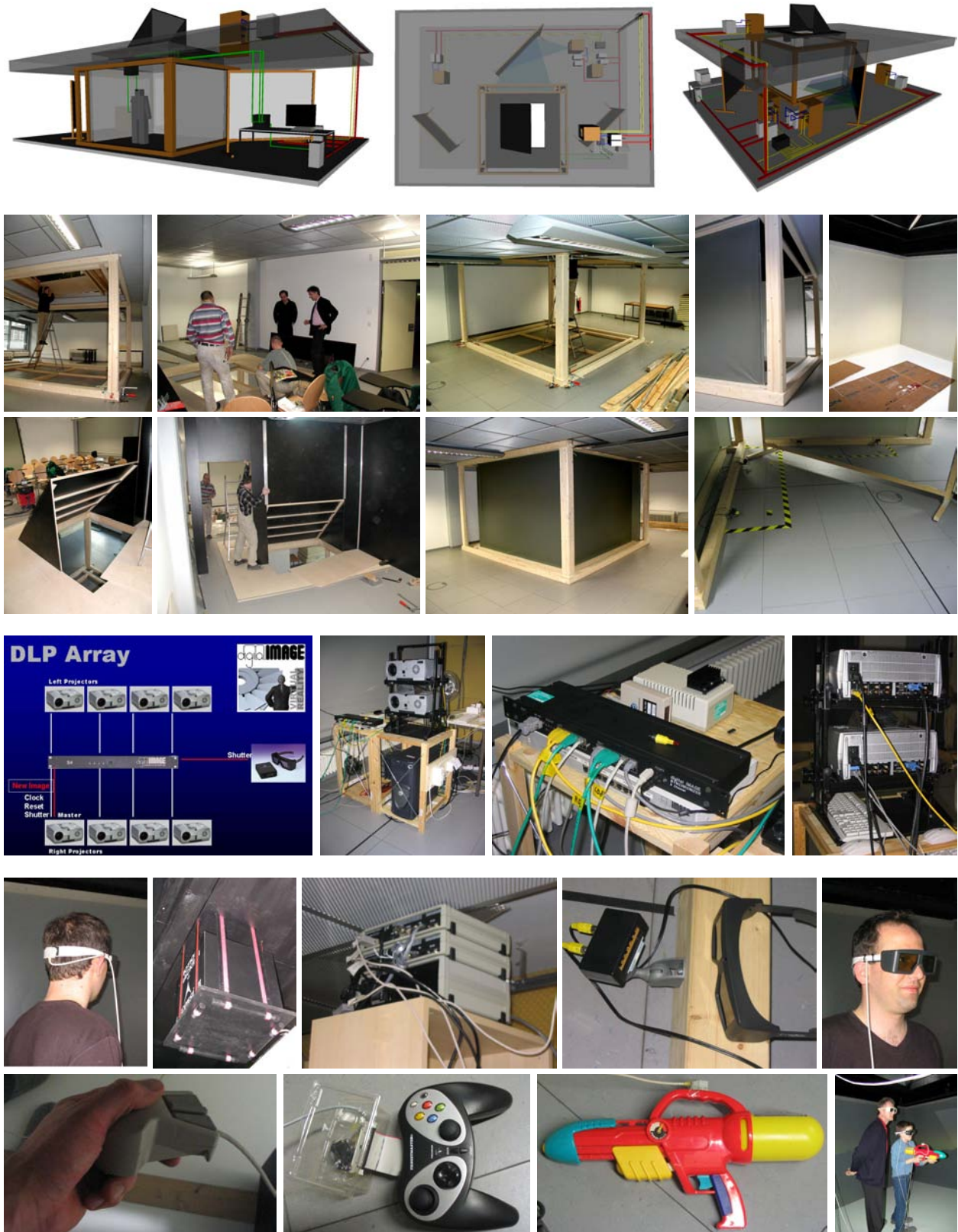


Figure 1.27: Physical setup of the *Definitely Affordable Virtual Environment* DAVE. Row 1: Schematic overview with two levels of synchronization (yellow,red) and tracking system (green). Rows 2,3: I had to plan the physical installation – an exercise in elementary geometry – and to survey the execution: the wooden frame, built and installed by a local carpenter, the screen (one piece, 9×2.25 m) made in England, and the thin-film mirrors from Munich. Row 4: The magical projection from digital image, four pairs of projectors and the synchronizer. Rows 5,6: Head tracking with sensor (5a), emitter (5b), and bird units (5c), shutter glasses, and 6DOF input devices.



Figure 1.28: Initial problems: The DAVE is a prototype. The magnetic field is ‘dragged’ through the hole in the ceiling (1a,1b). Kai and I take $9 \times 9 \times 6 = 486$ measurements in a 25 cm grid for use with the calibration library libTrCalibr2 from Utah. The projection screen starts to tear in the corners and I have to reinforce it (1b-1d). Color bits are spread unevenly over 1st and 2nd turn of the color wheel (2a). Lamps explode until we change to digital image’s open projector case (2b-2d). Torsten adds fast fuses to protect the other projectors when a lamp explodes.

Does it always have to be a Cave? From an engineering and economic point of view a Cave is just a tool, and using it has pros and cons. Experience has shown that immersive 3D has tremendous advantages over other types of 3D technology with (i) 3D data with a very complex spatial distribution, and (ii) in situations where precise controllability of virtual tools and devices is important. The spatial understanding of a complete car engine or a swiss cheese can be greatly enhanced by the sense of motion of the own body – especially when the head can literally be stuck into the 3D model. Only in a Cave one can walk *around* an object floating in space. – The same level of immersiveness is in principle also possible with portable 3D devices like cyber-helmets, where a pair of small built-in monitors are integrated with the glasses. They may require less hardware and (some day) provide the same quality, but with respect to software they pose the same problems.

What does ‘virtual reality’ really mean? The possibilities of immersive VR go beyond adapting desktop software; it is a new quality of media with its own very particular rules and perspectives. A central notion in this context is *interactivity*. The limitations caused by insufficiently rich 3D interaction become particularly ardent in a Cave. As was pointed out in section 1.2 there is the fundamental dichotomy between modeling and viewing. The pre-processing in between destroys the way back to the modeler as it removes, e.g., the modeling history. Without modeling capabilities in the viewer interactivity is essentially limited to changing transformation matrices to move objects or the user (navigation). This is VR only in the sense that a non-existing surrounding environment is displayed. However, it is not a huge step beyond, e.g., presenting a drawing or a painting (not: a photo) in an art gallery. This type of VR is basically a walkthrough-painting.

Beyond walkthrough paintings: Technical foundations for responsive VR. Cave technology cries for dynamic interaction: *changeability* is the greatest asset of computer-generated virtual environments. VR knows practically no restrictions in terms of construction cost or amount of building material, and phantasy may even ignore the laws of physics. *Any* idea can be realized – as long as there are practical ways to express and communicate these ideas to the system.

Note that a Cave user has no keyboard; to type in a name is difficult, as well as to note down something. A Cave has no force feedback. Users find it difficult to, e.g., draw a straight line in free space, instead of on a table as one is used to. Nothing prevents a VR visitor from penetrating the ghostly projected objects around him. Floating 3D objects have no weight at all. Physical properties can only be suggested with graphical means: With changing colors, with objects snapping to positions, with particle systems, with shadows and projection on auxiliary planes, with dynamic connecting lines, and with gizmos as supplemental supporting objects. They are set into the scene on demand as a ‘physical’ representation of a piece of information or an action that can be performed.

On the technical level this draws on the integration of modeling and viewing; on ways to inter-twine the inevitable pre-processing with the modeling; on the preservation of semantic information through the whole chain; on re-parametrization to identify and isolate the essential DOFs; on powerful construction tools on a high-level of abstraction. This requires all the technologies developed in the contexts of DL (ModNav/ V^3D^2) and cultural reconstruction (Charismatic, Epoch).

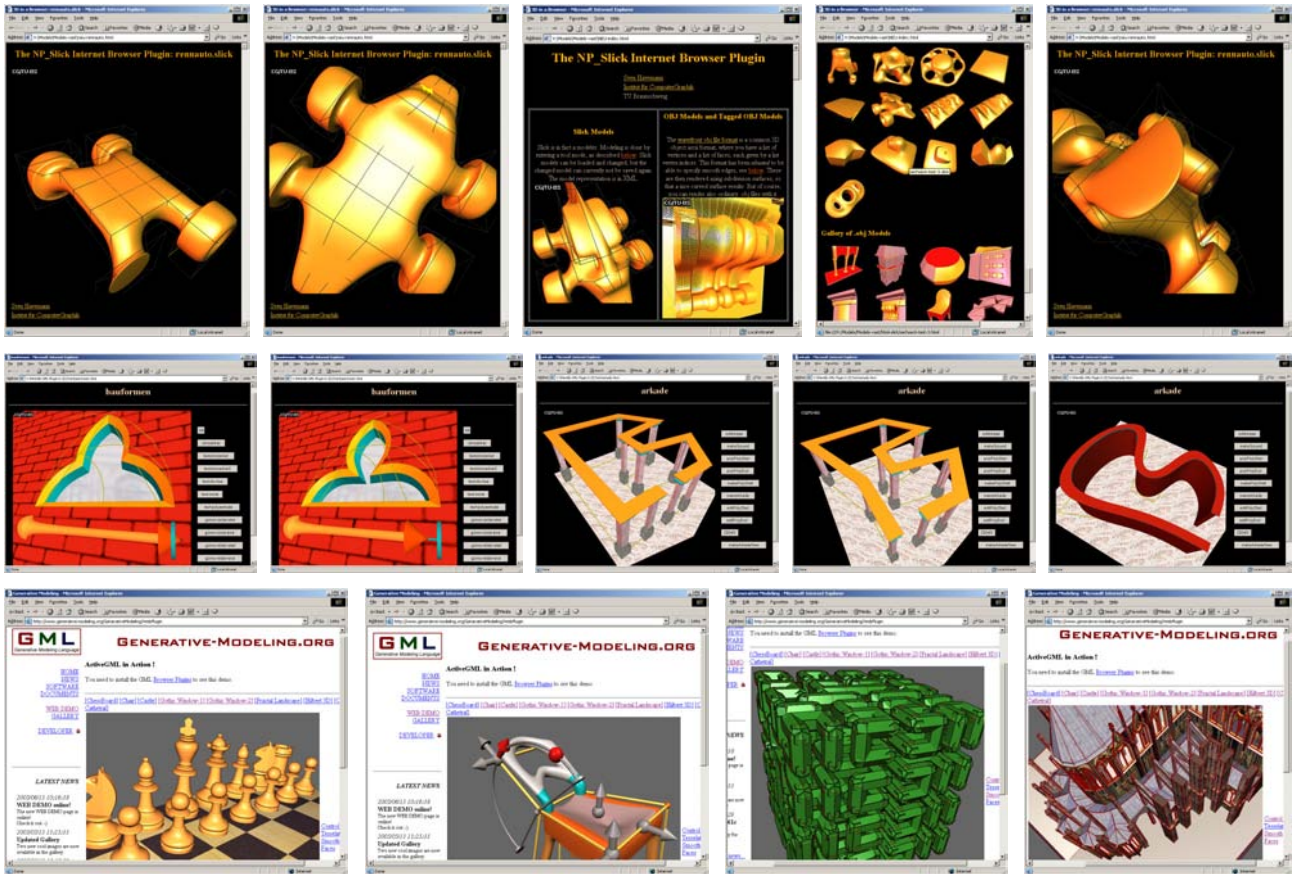


Figure 1.29: Three generations of web plugins for interactive modeling and adaptive visualization.

3D Modeling in a web browser: Meaningful interaction for GUI-less modeling. The described technical requirements for responsive VR in an immersive environment are remarkably similar to the requirements in a totally different setting, namely for web-based 3D modeling. The main parallel is that internet users do not expect a web interface to have any complicated GUI elements or deeply nested menu hierarchies. HTML forms are fairly limited compared to a fully-fledged GUI toolkit like Qt. So when trying to realize 3D modeling, in the sense of shape design, in a web browser, this needs to be accomplished with a fairly minimalistic user interface. Best is to represent actions directly in 3D.

René Berndt and I made our first experiences with combining web technology and 3D modeling in 2001. Since then we have produced the three generations of web browser plugins shown in Fig. 1.29. The first plugin (row 1) used combined B-rep technology and offered the two ways of modeling shown to the left and right in 1.29 (1c):

- **Slick modeling:** The modeling functionality (from Figs. 1.19, 1.20) was mapped to a GUI-less browser plugin by making heavy use of key strokes. Pre-defined high-level tools are activated by pressing, e.g., the e-key for extrude. Picking applies the tool, pressing 1-4 activates the ‘parameter adjustment’ mode of mouse movements.
- **Edge sharpness annotation for .obj models:** The edges of a combined B-rep are sharp by default, so that models exported from a CAD program such as Sketchup (section 1.2.2) are by default polygonal. The plugin allows to select and smoothen edges and faces of a model read from a .obj file. Also a whole set of faces can be selected and smoothened or sharpened, or just the edges on the boundary of this region, or just edges *emanating* from it.

In any case the modeled or annotated 3D shape can be put into a database to produce overview pages like 1.29 (1d), from which a model can be selected to adjust some of its parameters (tail angle in 1e). The interesting technical details of this approach have unfortunately never been documented.

The second version of the plugin (row 2) used the first version of a much more general modeling technology, the GML. It was first shown at the final review of the Charismatic project and has since then matured into the solution that will be presented in the following section – and in detail in the rest of this thesis. With Internet Explorer 6, Microsoft ceased to support Netscape/Mozilla plugins (which it had before) so that we started to develop in parallel a version of the plugin as an ActiveX control. The resulting *ActiveGML* plugin (row 3) was presented by René in 2005 on the Web3D conference [BFH05]. The latest version of the plugin is always available from the GML homepage www.generative-modeling.org.

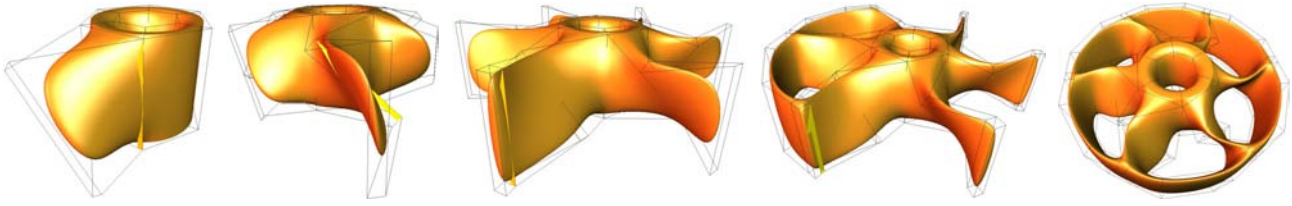


Figure 1.30: Turbine made with *Slick*. Such repetitive modeling tasks immediately suggest a stack-based approach.

1.6 A New Type of 3D Technology: The *Generative Modeling Language* GML and its Four-Layered Software Architecture

One of the most serious objections to the Python and *Slick* toolkits is *controllability*: A CAD-quality model, e.g., a car engine, cannot be designed with these toolkits. Both toolkits are also seriously affected by the persistent naming problem, which takes in this case the form of the *picking problem*: Selection is done by picking and ray intersection. The picking rays are even conserved in the model file (see Fig. 1.17). The *Slick* toolkit, in terms of modeling capabilities more restricted than the Python-based toolkit, was nevertheless the trigger for developing the GML. First note that a model is created from a sequence of tool applications, repetitively issuing these three steps:

- **tool selection:** from a menu or with a shortcut key a modeling operation is selected
- **tool application:** by picking the mesh a vertex within a face is selected and the tool is applied there
- **parameter adjustment:** the GUI contains four $[-1, 1]$ sliders to enter at most four tool parameters

The simplicity of the toolkit, and its distinct shortcomings, immediately and clearly suggest methods for improvement by generalization: Modeling acts often locally, i.e., the next operation B is applied very close to where the previous operation A was applied. The mesh locations are highlighted with little half-arrows a and b , so-called mesh iterators. Since mesh iterators can crawl over the mesh it is very natural to wish one could tell the computer how to generate b from a , rather than performing two mouse clicks to pick a and b as independent mesh iterators. This is especially annoying when a and b are equal: In this case one would like to have a way of saying ‘take the halfedge you already got’.

A very similar desire arises with respect to the parameter adjustment. The *Slick* gallery in Fig. 1.20 contains objects that exhibit strict regularities, for instance the turbine propeller (1e). Each of the turbine blades is made from three extrude operations, as illustrated in Fig. 1.30. This operator has a rotational twist parameter; when it is $+0.12$ with the first extrude it should be -0.12 with the second, so that the face has then again the original orientation (1.30 b). Since the parameter sliders stay where they are, the most efficient way to create a propeller is to perform first, for all blades, the $+0.12$ extrusions (one pick per blade) and then the -0.12 extrusions with another pick per blade. Then the third extrusion is performed to finally connect the blades with tunnels to create the outer ring (1.30 c-e).

What one would like to do instead is to carefully design the three extrusions of the first blade to then simply repeat them for creating the other blades. This would permit to change the number of blades more easily. Second, one would like to multiply the 0.12 by -1 rather than enter -0.12 . Third, when the model is finished one would like to have one slider to change all 0.12 to 0.14 *at once*, which should automatically change all the -0.12 to -0.14 .

A stack as a simple solution that leads to –PostScript. Problems of this type can be very easily solved when the result of an operation, e.g., a newly created face, is put on a *stack*. To process it the next operation can then simply fetch it from the stack (*pop* it). The result of this processing is then pushed again back on the stack etc. This way it is very easy to set up and define an *assembly line* of operations that are sequentially applied to a given piece of initial data. Note that it is sufficient to issue only the operations one after another; parameters are passed quietly over the stack, without the need to explicitly declare any input- or output variables.

This and other examples quickly raised the attention on stack-based languages. One of these languages has great importance in the field of 2D graphics as a page description language for desktop publishing, namely Adobe’s *PostScript*. It is not well known that *PostScript* is a programming language. After the first 25 pages of chapter 3 from the *PostScript Language Reference*, also known as the *PostScript Redbook* [Ado99], made it very clear how to write an interpreter for the language core. The result was, right after christmas 2001, the first version of the GML interpreter.

The original motivation for developing a formal language backend was only to have a *notation* for GUI automatization, i.e., for the automatization of tasks that are performed with a GUI. After all the objective was to render interactive shape design more efficient. But the language itself has proven so fruitful that the attention has shifted a bit from interactive design to the general shape description problem. It turned out that a stack-based approach is surprisingly well adapted to 3D graphics as many elements of the language can be directly mapped to concepts from graphics.

```

1: /stdCyan setcurrentmaterial
   (0,0,-2) (1,1,0) 2 quad
2: 5 poly2doubleface
3: (0,1,1) extrude
4: (0,0,1) (1,0,1) normalize
   0 project_ringplane
5: (2,0,0) (0,1,-1) 2 quad
6: /stdYellow setcurrentmaterial
   5 poly2doubleface
7: 0 bridgerings

```

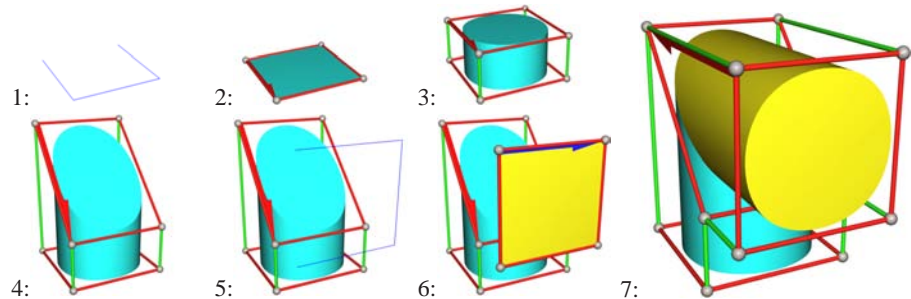


Figure 1.31: GML example for modeling with a stack-based language. 1: Parameters of the quad operator are the midpoint (0,0,-2) and extension (1,1,0) of the quadrangle and 2 as a mode flag. The quadrangle is put on the stack as an array of four points. 2: The polygon is converted to a mesh face, pushing a halfedge. 3: The extrude operator expects a halfedge and extension vector, and pushes the halfedge of the resulting face. 4: The face is moved by projecting it in z -direction (0,0,1) onto a plane. 5,6: A second quad face is created. 7: The quad faces are bridged with smooth edges. Thanks to combined B-reps the red/green mesh is sufficient to define the rounded pipe.

GML Simplicity: All functionality comes from the operators. The GML has, like PostScript, only two higher-level built-in data structures, arrays and dictionaries. An array of 3D points can be used as a simple polygon, a *path* in a mesh is just an array of halfedges. A dictionary can be used as a tools library and also as an object of a fully-fledged class in the object-oriented sense, or just to represent a dialogue box. Halfedges and 2D and 3D points are atomic – *literal* – data types in the GML like integers and reals. The set of these basic types is even extensible.

The GML interpreter has, like a Postscript interpreter, no built-in functionality or any reserved keywords. All of its functionality comes from the operators. Operators are organized in libraries that are registered at runtime with the interpreter. The GML does not have PostScript's extensive set of operations for typesetting and 2D graphics, but it has several libraries for 3D modeling, meshes, polygons, vector math, interaction, etc. A small, illustrative example of 3D modeling with the GML is given in Fig. 1.31. It shows how a piece of data (a quad) is created and processed, i.e., transformed (extruded, projected), and then another piece of data is created (another quad). The result is *two* objects on the stack, which are then both popped and processed by a combining operation (a bridge or tunnel).

The GML is, like PostScript, a language that is so simple that it does not even have a syntax or a grammar that are worth mentioning. The whole language can best be described by one pageful with a dozen rules, which are listed in Fig. 5.5. The GML parser is actually only a lexical scanner. It converts ASCII source code to a sequence of tokens, indeed implemented as a `vector<Token>` on the C++ level. To execute a GML program then only means to execute this sequence to token by token, each time performing the appropriate actions: to put a value on the stack or to call an operator. Quite remarkably the GML is in fact also a *functional* language, simply because a function is nothing but an array containing operations – and functions can also be treated like arrays.

Related Approaches. As we found out later, by pure coincidence, there is another stack-based language for 3D computer graphics that is called GML. It is used as a scene description format for a raytracer. To actually implement it was the semester project CMSC 23700 [RDR03]. The students were only given its formal description and its desired behaviour. – It appears that it turned out to be useful since it was later also the basis of a raytracing contest.

Furthermore in his *Notebook* column in IEEE CG&A, Andrew Glassner has developed a stack-based language as a tool to describe *Crop Art*, interesting spirograph-like 2D patterns [Gla04]. It is not so clear, though, why he did not simply use PostScript. – It appears that stack-based languages have been used due to their great simplicity at a number of occasions also by other people for rapidly creating small tools.

Despite their success in 2D, stack-based languages have never gained much attention as tool for describing 3D content. This is all the more surprising since 3D graphics can benefit much more than 2D from a data format that is also a programming language. The main content in 2D desktop publishing is text, which has very few exploitable dependencies. PostScript's capabilities are not even used where they would make sense: To use PostScript only for the print-out of an Excel sheet is a waste since it is *easily* capable of representing the sheet itself, i.e., the maths behind and the relations between spreadsheet cells. The ghostscript interpreter, for instance, might be fed only with the values of the essential cells from which the rest of a spreadsheet can typically be produced.

The fact that the GML can actually compute values, has loops, and can perform conditional decisions, is much more important for 3D than 2D graphics: Every complex geometric shape, be it a building or a mechanical object, results from assembling several individual parts that are highly dependent on each other, simply because they need to fit together.

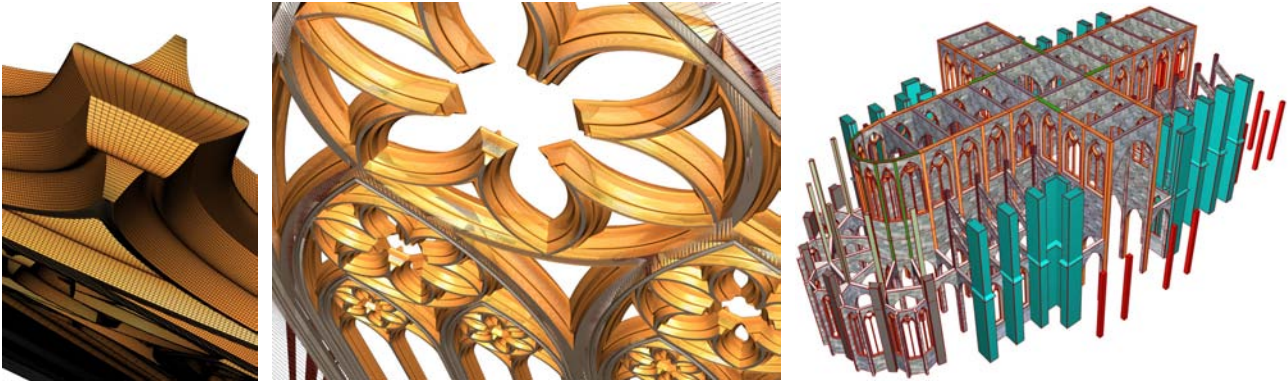


Figure 1.32: Gothic architecture example. Window tracery is an amazingly rich but challenging domain for parametric and procedural design. (a) Profile detail of the main rosette at full resolution with subdivision depth 4. (b) Wireframe of the complete Gothic window with two levels of recursion, sub-windows and sub-sub-windows. (c) The *Procedural Cathedral*, a raw building of the Cologne cathedral with about 70 windows, here simple style. All geometric elements are created by appropriate functions: Objects become operations. 12 KB compressed GML

1.6.1 Four-Layered GML Software Architecture and Thesis Overview

It is important to emphasize that the scope of the paradigm change in 3D technology proposed in this thesis is not limited to representing 3D content by a stack-based language. This alone will not help at all. There is sometimes the misconception that the GML is only a formal language; but equally important is its runtime engine. So the more appropriate label for this technology appears to be the term *GML framework*.

The arrangement of the four software layers follows the principle of information unfolding (Def. 1.1). High-level information is on demand successively unfolded to produce larger amounts of more explicit derived data. These data are cached only as long as they are needed since they can be re-generated. The following explanation begins at the low level, corresponding also to the order of the chapters. The broadening of the view from local to global is illustrated in Fig. 1.32.

- **Level 1: Adaptive display of curved surfaces** – **Catmull/Clark patches** **Chapter 3**

On the lowest level shape boils down to triangles. To synthesize enough detail for close-up views (1.32a) a curved surface is partitioned into small patches that (i) can be tessellated quickly with a small computational effort and (ii) have a multi-resolution tessellation. This is vital to maintain an interactive frame rate also for overviews (1.32b). Patch setup, computation, tessellation and display is all only *local* – an important precondition for selective updates.

- **Level 2: Primitive-based shape representation** – **B-rep meshes** **Chapters 2 & 4**

The shape of a subdivision surface is completely determined by the much leaner control mesh. Mesh entities, the vertices, edges, and faces of the B-rep, are annotated with tessellation data. This are not only the Catmull/Clark patches for smooth faces from level 1: Faces may also be polygonal, and then the face carries (a reference to) the triangulation of the face boundary. All tessellation memory is returned to a global pool when entities are deleted.

- **Level 3: Shape manipulation operators** – **Euler operators** **Chapters 2 & 4**

Although B-reps are a primitive-based representation they can also be created procedurally with Euler operators. They have some important properties: They are *closed*, so they do not invalidate a valid B-rep, they are *sufficient*, so every B-rep can be created with them, and they are *invertible*, which is the basis for undo/redo. Every B-rep can be created by/converted to an Euler sequence, regularities in the sequence can be exploited for semantic LOD.

- **Level 4: Formal shape description language** – **GML** **Chapter 5**

As soon as shape can be created with operators, specific operator sequences can be grouped together as re-usable shape construction macros. A simple formal language is essential as an explicit notation for these macros. The structure of the GML is very well suited to representing the assembly-line style of modeling in interactive shape design. At the same time it permits to exploit the similarity between shape design and programming.

All levels of the GML framework are deliberately based on well-established, well understood standard technology. B-reps, Catmull/Clark surfaces, Euler operators, and stack-based languages are not at all new. The innovation lies rather in the novel way of combining these components: Nobody believed it would make sense to create meshes with PostScript.

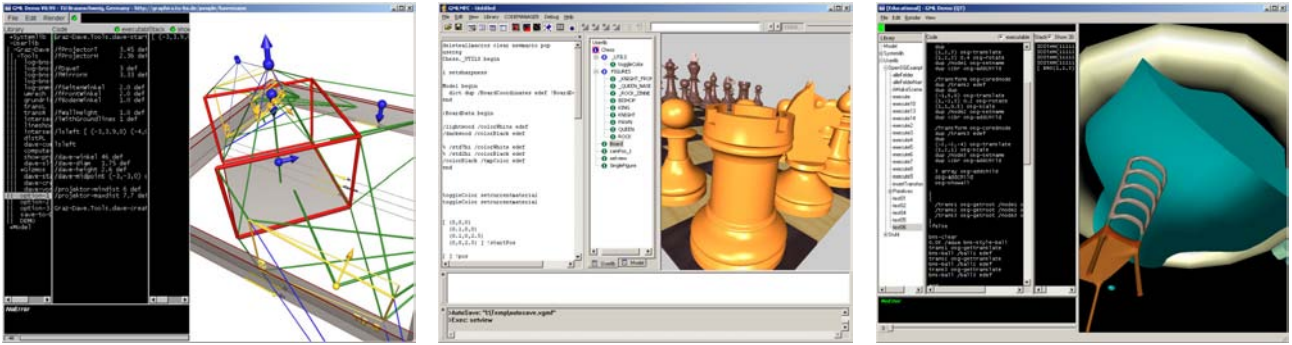


Figure 1.33: Different available IDEs for GML. The software design of the GML is that of a plugin. The GUI code is just a thin layer on top of the GML interpreter and the *Meshlib*, which permits for a rapid adaptation to, e.g., the *Fast Light Toolkit* fltk (a), the MFC framework from Microsoft (b), and Trolltech's Qt (c).

1.6.2 Features of GML-based 3D Technology

The term *Generative Modeling* reflects an alternative paradigm for representing shape, namely to understand a 3D shape as the result of a sequence of operations, the *shape generating functions*, rather than a static set of low-level shape primitives. The layered GML architecture from the last section is an attempt to operationalize this view on shape. Not all implications of this technology may be obvious; the following list shall therefore highlight a few of them. Note that some of the arguments refer to details of the framework that are only explained in the following chapters.

- **Paradigm shift from objects to operations**

Throughout the nineties, from TBag over VRML to OpenSG, were 3D shapes often understood as objects in the object-oriented sense: An object of type Cylinder has member variables height and radius which are floating-point numbers. But this view completely neglects the procedural aspect: All real shapes are the result of some manufacturing process; synthetic 3D objects are created using modeling tools; in other words, shape is in many if not most cases the result of applying certain operations.

Objects can in many cases also be understood in the functional sense, as operations. A node in a graph can be identified with the operation of creating this node; an edge can be seen as the action of connecting two nodes. The advantage is that then a graph pattern can be formulated as a parameterized graph weaving operation. – This explains how meshes can be created with the GML as efficiently as whole scene graphs.

- **Shape design as rule design: Plugging rules Lego-like together**

The reader is urged to try an experiment: Ask somebody nearby to place a dozen dots on a sheet of paper. Repeat the experiment with other simple shapes like triangles, quadrangles, and circles, or ask a person to draw an arbitrary continuous pen stroke for thirty seconds. The result is surprising: Almost everybody will immediately invent some sort of rule. Depending on the person's creativity he or she will place the dots in a row, make a grid of quads, an alternating sequence of triangles, or draw a nice meandering stroke.

Assembling GML programs is a little bit like playing Lego with compatible rules instead of compatible bricks: Output data from one operation are input data to the next. The greatest challenge in operator design is to define the input signatures (the order of the input parameters) in a way that permits the most flexible combinations of processing chains.

- **Procedural models deserve a procedural representation**

The generative method works best if many dependency relations exist between shape data. The ideal case is when a few initial parameters of a geometric construction are sufficient to derive a complete building or a complex machine part. The GML supports in an ideal way the reduction of parameters. A GML program is just a sequence of tokens, and executing the program means to execute each token in a row. This permits to easily and transparently replace concrete values by variables, and vice versa: '1 2 3' and '1 x 3' are absolutely identical (modulo side effects) if x is set to 2, or if x is an arbitrary function that eventually pushes the number 2 on the stack ('evaluates to the number 2').

- **Bridging the gap between modeling and viewing**

The GML together with its runtime engine can be seen as a modeler with an optimized realtime renderer, but also as a 3D viewer with an integrated modeler. This modeler is very general since with its Euler operators it is low-level but complete. Correspondingly a GML model can be much more than a conventional mesh model, it may even integrate a customized modeler to create an unlimited manifold of similar models. An example is the *Cave Configurator* in Fig. 1.33, where only a few high-level parameters can be adjusted with the blue arrow gizmos representing 3D sliders.

- **Polygonal and free-form geometry within the same data structure**

Combined B-reps realize the generative principle already on the data structure level: Subdivision surfaces are generated procedurally from the control mesh by simple but powerful recursive refinement rules. Smooth free-form surfaces can be created with minimal effort just by switching the sharpness flag of a few edges in the mesh: Every face that has at least one smooth edge is rendered as Catmull/Clark subdivision surface. This isolates just the *essential DOFs* for shape design: The complexity is reduced by more than two orders of magnitude because one quadrangle face of the control mesh can unfold to as many as 256 quads of the tessellation.

- **On-line multi-resolution mesh modeling**

Conventionally a static shape is converted to a multi-resolution mesh in a pre-processing step. Our approach intertwines this pre-processing with interactive display. The actual tessellation of a mesh face, irrespective of its type, is generated only on demand, i.e., when it is requested for display. In case of local mesh modifications the tessellation is *selectively updated*: The data structure keeps track of allocated, deleted, and modified mesh entities by means of a *touching scheme* (see section 4.3.2). The tessellation is also incremental: Surface resolution can be traded for interaction speed using a *quality parameter*. This means that the tessellation is organized in a way that it is *refineable* so that the computational cost can be spread ('amortized') over a number of frames.

- **Preservation of model semantics throughout the whole processing chain**

The entire knowledge about which mesh entities belong to which part of the model is preserved on the level of individual half-edges of the control mesh. This means that the complete way through the unfolding of information in levels 1-4 from sec. 1.6.1 is recorded. Picking a Catmull/Clark patch with a mouse click one finds the half-edge the patch belongs to. The half-edge carries the ID of the Euler log record of the unique operation that created it. The log record belongs to an Euler macro. And on the macro level it is possible to identify the parts of the model by navigating through the macro hierarchy on the GML level, traversing the macro parent/child relation. The macro resolution can be defined by the user. The whole mechanism is even fast enough to let faces of the model be used for interactive sliders.

- **Generality through extreme simplicity: The GML is extensible**

The GML framework has only very moderate requirements for a surface representation or modeling method to be integrated. The interface is *extremely* simple as it requires only to define a number of operators: They need to pop input parameters from the stack, complain if there is an error, otherwise process them and push the result back on the stack. In a few *Studienarbeiten* this approach was applied to integrating implicit surfaces, convex polyhedra, and *3D Powerpoint*. It is a inspiring exercise to try and define, for a given application purpose, an 'orthogonal', i.e., minimal, set of operators, and to maximise their combinability by defining their signatures appropriately.

- **Generalization of modeling history**

One of the great things from the popular modeler *3D Studio Max* from Kinetix is the *modifier stack*: All modifications applied to an object are visibly stacked on another in the GUI. The parameters of each modification, e.g., the frequency of some noise applied to the object, can be changed at any time. The GML generalizes the modifier stack as it allows to treat and combine any sequence of consecutive processing steps as a single high-level tool. Because of the persistent naming problem the modifier stack of *Max* is frozen when an object is converted to a polygonal mesh and low-level mesh operations are applied. The GML can record operator sequences on the half-edge level, of course.

- **Generalization of data flow graphs**

The GML is also capable of representing processing steps arranged in a dataflow network. Formally this is a directed acyclic graph (DAG) where the output of one operation can be fed into the next. For every DAG exists a partial linear ordering. In the simplest this can be directly turned into a GML program by simply concatenating the names of the respective operators in this order, separated by whitespaces; possibly a few calls to *dup* and *pop* must be interspersed. Since the GML is a functional language it is more general in that it allows operators, i.e., processing nodes, also to be parameters. As it furthermore allows to collapse sub-parts of the network into a single combined operator (node), it even permits to send sub-networks as parameters over the connectors within the network.

- **Solution to the annoying code generation problem**

GML inherits from PostScript a unique feature. PostScript is the *invisible language*: In terms of *quantity of automatically generated source code* PostScript is without a doubt unparalleled by any other programming language. Whenever a document is printed on a PostScript printer, the printer actually executes the document as a computer program written in PostScript that, as a side effect, produces the bitmap that eventually appears on a sheet of paper. The vast amount of available PostScript drivers for all kinds of software indicate that it can not be immensely difficult to generate valid PostScript code. PostScript is ASCII. It is very interesting and instructive to have a look at the result produced by AutoCAD when converting an engineering drawing to PostScript.

An often-heard objection to stack-based programming languages is that they are horrible to program ('stack acrobatics'). The answer to this is that *nobody wants to program*: A stack-based language was chosen to get rid of coding altogether.



Figure 1.34: Separation of structure from appearance. A generic chair model, parameterized with five 3D points, is adapted to given chairs. The arrows are slider gizmos to move the five control points interactively, so in fact they re-parameterize the DOFs. 12 KB GML code. Note that surprisingly different objects share the same structure.

1.6.3 Potential: A Wealth of New Questions

The *really* difficult problems are not solved in this thesis, of course. The considerations from the last section nevertheless permit to hint at possible directions for further research in response to the tough issues that were mentioned in section 1.4: The proposed change of perspective may after all assist in finding adequate solutions one day.

- **The semantic gap between a shape and its meaning**

The example from section 1.4 for the semantic gap between a shape and its meaning is the scanned amphora. This refers to the *inverse* problem of shape synthesis, namely shape recognition. What does it mean to recognize a shape? This question is flagged below as a possible area for further research.

- **Solution to the modeling bottleneck**

Generative design with its possibility to specify rules for automated shape creation, instead of creating individual shape instances, is certainly a mandatory first step to increase the modeling efficiency; however, only this is not yet sufficient. What is still lacking in the current thesis is an efficient method to design shape creation rules interactively; and this will be the difficult part.

- **Digital libraries of solutions to modeling problems**

Re-using the solutions to modeling problems that have already been solved before is possible only when these solutions were formulated in a generic, re-usable way. The operator calculus offers in principle the possibility to do so, and in fact a substantial library of auxiliary functions are part of the GML model of Gothic architecture ([HF04a, HF04b], also see section 5.4.1). But again the hard problems are those of maintenance (versioning, consistency) and the classical digital library services: *markup*, *indexing*, and *retrieval* of construction rules.

- **Extremely compact shape representation**

The size problem is indeed solved by procedural modeling. Quickly gathering the GML files for *all* models in this thesis, including some redundancies, yielded about 1 MB of source code – which is only 175 KB zip-compressed.

- **Dynamic virtual worlds**

The most important ingredient is the integration of full modeling capabilities into the viewer – which has been achieved. The GML permits for arbitrarily complex responses to user events. Powerful callback functions can be attached to any piece of geometry in the scene, callback functions can even be interactively assembled and defined at runtime. – Again, however, this is only the technical prerequisite: It still remains to find ways for a meaningful interaction/communication with a virtual environment – efficient ways for humans to express their creative ideas without much frustration.

- **Using 3D ‘out of the box’**

This is solved by three measures, (i) the integration of modeling and viewing, (ii) the possibility to isolate the essential DOFs, and (iii) the design of the GML as a plugin: Executing ‘1452 castle’ might create and display a typical castle from 1452 in any third-party application that provides an OpenGL context, provided a suitable castle library is provided.

A wealth of fruitful new questions should be the result of solving some of the annoying questions mentioned above. The following list is a preview to the collection of a dozen areas for *Future Work* at the end of the thesis in section 5.6.

- **GML as a general exchange standard for procedural models**

The great goal is an architecture that allows to exchange intelligent 3D components between different software systems so that they still remain high-level editable in the target system. The GML is intended to be a ‘smallest common denominator’ for the description of procedural models, analogous to triangles for the description of surfaces. This claim remains to be proved, of course. It can be formulated as the *GML efficiency conjecture*:

Most if not all concepts in today’s state-of-the-art parametric and procedural modeling systems can be *efficiently* represented using a stack-based operator calculus like the GML.

Examples already mentioned include the construction history (modifier stack) and the dataflow graph; further candidates are dialogue boxes and high-level primitives, scene graphs, keyframe animations, articulated bodies, particle systems, soft bodies, and procedural shaders, which may all be very well captured with operators.

- **GML as shape markup language**

GML+OpenSG may be a valid successor of VRML/X3D. History showed that to standardize only the language is not a solution. Without a standard runtime engine inter-operable 3D components will remain a myth. The power of GML as a markup language will be shown in section 5.6.12. There are ways to translate an X3D file (=VRML+XML), as well as other legacy formats, to GML syntax automatically (see Figs. 5.61 and 5.60 in section 5.5.3).

The great advantage of such a transformation is that it removes the artificial separation between markup language and processing language. Both are replaced by a single extensible language for static as well as procedurally generated and/or manipulated shapes, and one extensible standard runtime environment – as opposed to the fragile combination of VRML/X3D + Java/JavaScript + proprietary viewer. This unified view has a great potential: Hierarchical Euler macros can for example be seen as the continuation of a scene graph below object level, etc.

- **Shape configurability for mass customization**

Mass customization is a huge trend: Market pressure, shorter product cycles, and simply the advantages of digital technology lead to the implementation of a purely digital workflow in many industrial domains. With computer power comes customizability: A CAM/CAE machine does not have to produce only identical objects in the same production line; in one word, it introduces *changeability*.

The configurability of shape is thus less of a problem for the manufacturing process, it is more an interface problem. Questions are how to author, configure, store, and transmit changeable shapes reliably, how to let non-expert users edit shape, and how to suitably limit the degrees of freedom to avoid infeasible input. The most serious argument *against* end user 3D shape editing is *cost*: Creating masses of 3D user interfaces is simply too expensive. – Needless to say that the GML framework is qualified in a unique way to be a solution to the interface/exchange problem.

- **Tangible 3D: Truly responsive virtual worlds**

A *gizmo* is a 3D object that is artificially put into a scene to represent an operation, such as a parameter change or a switch. The vision of *Tangible 3D* is to operate gizmos with bare hands, which are the ultimate form of *human-computer interface* (3D-HCI). In the recent science-fiction movies *Matrix* and *Minority Report* computers are controlled by pointing with bare fingers at transparent 2D interfaces freely floating in space. A less far-fetched and more concrete usage scenario are 3D shopping terminals. They may be thought of as a generalization of the familiar selling machines with a small 2D screen, such as ATMs (for drawing cash money), or terminals that sell railway tickets.

This leads to two fields of research that are tightly related: First, to develop the right input devices, and second, to identify suitable interaction metaphors and devise general design guidelines for responsive 3D applications. Our first results, together with Hyosun Kim and Georgia Albuquerque, of un-instrumented 3D interaction with camera tracking of finger tips are very promising [KAHF05]; but much, much more is possible – and will be done.

- **Update-able shape data structures: The quantitative increase in speed requires new a quality in algorithms**

A quantitative increase, e.g., in processing speed and available main memory, can turn into a qualitative change simply because suddenly things become possible that were not possible before. Any sort of pre-processing becomes obsolete when it takes longer to read the processed data from a file than to compute them. But then the users will sooner or later also want to *change* the input data on the fly.

Many existing methods in computer graphics, like *raytracing*, use to operate in a monolithic input→algorithm→output fashion. They need to be completely re-arranged when used interactively: (i) the processing is quickly performed only partly to produce a rough approximation, (ii) the output data are multi-resolution, and (iii) is recorded which output data B_j are produced from which input data A_i . When A_i changes to A'_i then (ideally) only B'_j must be (re-)computed, and not the complete result $B_0 \dots B_m$. – An example for such bi-directional caching over several levels are the (*progressive*) *combined B-reps* (sections 4.4,4.3). The same is possible for other surface types, e.g., for the tessellation of metaballs.

- **Logging and re-constructibility: Re-ordering construction sequences**

A naive solution to the problem of authoring procedural shape families is *logging*. With a macro recorder, for instance, all user interactions and events are captured, they can be repeatedly played back, maybe even some sort of editable code is generated (Maya MEL, Microsoft VBA), etc. It makes no sense, though, to record all slider ticks during parameter adjustment. Similarly, interactive modeling involves much trial and error. So there must be a way for the user to tell the system which steps are essential and must be recorded, and which are not. Even more desirable is, of course, to have a system capable of deciding automatically what is important and what is not. With an operator-based calculus to formalize construction sequences this boils down to the problem to automatically optimize these sequences.

- **New research area: Comparison of shape constructions**

With a notation for construction plans at hand it becomes in principle also possible to reason about and compare shape constructions. Note that there are *many* different ways to build the same shape. This is not only a technical problem. It reflects the fundamental fact that every shape instance naturally belongs to different shapes families: A triangle is, for $n = 2$, an n -dimensional simplex shape, as well as a rough approximation to the circle.

More general than the question whether two shapes are similar is the question whether two *shape families* are similar. This more general question is not inevitably also more difficult to solve. Without imposing any restrictions it is, of course, undecidable [Tur36]. A reasonable approach is to define a *normal form* containing a limited set of building blocks for procedural constructions. One such building block is for example the linear sequence (Fig. 1.2). Imaginable is a small set of generic construction templates that can be tried out with a given dataset, e.g., a scanned stairway, to mimic its procedural construction; this should in this case also involve using a loop. Such a normal form with generic procedural building blocks might as well be of great benefit for the rapid design of shape families.

- **New research area: Automatic Shape Understanding as generalized shape matching**

The semantic gap between a shape and its meaning is not closed in this thesis. The example from section 1.4 is the scanned amphora and the observation that most important fact about it is that it is indeed an amphora. But how can it be explained to a computer what an amphora is? – This is exactly the shape description problem.

Existing methods usually try to recognize an unknown shape by comparing it to a number of known shapes by means of, e.g., measuring the deviation between surfaces (Hausdorff distance). This is a somewhat superficial measure, though. Much better results are possible for all kinds of recognition tasks with a guided, model-based search than with a blind search: A task like pose estimation from a video sequence is much faster and more robust when a deformable 3D model of a human body is available that can be fitted to the video. The GML can assist in the rapid production of domain specific shape templates.

A generative description of a parameterized shape template makes it possible to express *structural* similarity. A concrete example is the generic chair shown in Fig. 1.34. It has only very few input parameters, five 3D-points, which makes it possible to quickly adapt the chair template to any given (scanned) chair. Although the models do not match in the strict sense (Hausdorff distance), the ‘important’ properties of the target chair can nevertheless be matched – according to the sense of importance that was coded into the chair template. This is exactly the kind of flexibility that is needed for the extraction of semantic/structural information. The images in the lower row show that a garden chair, a sunbed, a sofa, and a bed (not shown) in fact share the same structure as a chair.

To decipher the abstract structure always means to throw away some information. From the structural point of view, this may be only artifact information. For other purposes, however, this may be exactly the valuable information; for quality control of the object’s surface for instance, or to find differences between supposedly identical objects. – An area for research is therefore to find ways how the generic structure of a shape class and the detailed surface of a particular shape instance can be integrated into a single representation – hopefully in a way that combines the strengths and mutually compensates for the weaknesses of both ways to represent shape.

- **New research area: Managing procedural knowledge**

Procedural knowledge is one of the most valuable assets of individuals as well as academic institutions and commercial companies. The ability to satisfy an order relies on the knowledge how similar tasks have been performed in the past. Thus the preservation of this knowledge is critical. Procedural knowledge takes many different forms, which makes it very hard to reason about it.

One of the main reasons for the enormous success of the personal computer in the 1980’s was a revolutionary killer application: the spreadsheet. Software such as VisiCalc [Bri04] gave average persons for the first time the possibility to perform complicated calculations easily and instantly. Within the framework, a great amount of flexibility is granted. For more general tasks, the user must leave the restricted framework and fall back on a more general method to tell the computer what to do, namely *programming*. The missing link between spreadsheets and programming is a technology for *all* users, i.e., also those with no programming skills, that

- permits to specify a process, i.e., a sequence of processing steps,

- allows to integrate different applications into one process,
- but does not require any literal programming, and
- demands only a level of expertise comparable to creating a spreadsheet.

The idea to use the GML with its simple operator-based notation for representing procedural knowledge in general, i.e., as a technology to provide this missing link, was presented at the *I-Know* conference in Graz 2005 [HF05b]. The proposal was quite vividly discussed.

Thesis overview. The goal of this thesis is to develop a framework for generative modeling with meshes as the principal surface representation. For the principle of information unfolding to work properly it is not only necessary to specify input- and output data and then to optimize the conversion. The input data should be given in a form that indeed supports an efficient conversion, and the output data should be arranged so that results which are not yet computed can be integrated later. As the GML framework performs this unfolding even across several layers, it is just as important as it is laborious to optimize all of these processing steps simultaneously.

To summarize the task, what needs to be done is the following:

- **Chapter 2 – Theoretical Foundation of Polygonal Meshes**

Clarify on the fundamental properties of the shape domain, in particular on the subtle relation between cell complexes, 2-manifolds, and polyhedral meshes. Find a suitable, convenient definition of a *mesh*. Determine an operator interface for creating and modifying these meshes, and clarify on the properties of the operations: Are they closed, complete, and invertible?

- **Chapter 3 – Subdivision Surfaces**

Following the idea of *patch complexes* concentrate on finding a suitable type of multi-resolution free-form patch. It needs to be compatible with irregular polyhedral meshes, and it should not offer too many DOFs to remain artistically manageable. As it is at the lowest level of abstraction, with maximum unfolding, its generation/setup, tessellation, and adaptive display need to be highly optimized.

- **Chapter 4 – Practical Meshes**

Operationalize the more theoretical considerations on meshes from chapter 2. This is an applied and sometimes very technical chapter, intended to be useful also for students. It covers issues such as template parametrization of mesh data structures, realtime rendering from view frustum culling to LOD computation, the exact definition of the interface to the Catmull/Clark patches from chapter 3, caching and selective updates, logging and inversion of mesh construction sequences, and guidelines for the implementation on mesh modeling tools.

- **Chapter 5 – The Generative Modeling Language GML**

The final chapter eventually presents the GML as the formal calculus on the top level. It provides a *complete definition of the language on a single page* (Fig. 5.5) and a short general introduction to useful programming and modeling concepts. This is then somewhat intensified and applied right away to an extremely rich but challenging, long-standing domain of procedural shape design, namely *Gothic architecture*. – The chapter concludes with a proposal of a dozen directions for further research and an epilogue (in German).

Chapter 2

Theoretical Foundation of Polygonal Meshes

The purpose of this chapter is to give a proper definition of a mesh. This thesis builds upon meshes as the principal method for representing the shape of three-dimensional objects. But it is not so apparent, nor is it easy to define, what a mesh actually is. There is a gap, and a number of subtle differences, between the well-defined mathematical foundations of meshes and the way meshes are used in practice. Considering their importance, it is worthwhile to have a closer look at what these differences are.

The exposition begins with a few basic facts from algebraic topology. For aesthetic reasons, it starts at the very beginning, by introducing general topological spaces. The conciseness of the mathematical formulation permits to reach the point quickly where cell complexes can be defined as a well-defined way to look at meshes. This is then compared to indexed face sets as the main device for using meshes in practice. The main problem that is identified is how a mesh is embedded in 3-space.

Algebraic topology provides the major foundation for the description of shape, and it forms one of the grounds computer graphics can stand on. The second foundation comes from another branch of mathematics: differential geometry. Both fields are tightly related, but they consider slightly different aspects of shape: While the focus of topology is a qualitative classification and a distinction between global shape types, differential geometers are more interested in local analytic properties of curves and surfaces, such as differentiability and curvature.

The main concept here, and also the link between both of these branches, is the *manifold*. It is a very general and broadly applicable conceptualization that permits to express a variety of different aspects related to shape. The main reason for its usefulness is its versatility: Manifolds are a device for understanding abstract, high-dimensional spaces embedded in even higher-dimensional spaces. On the other hand, they can also be very concrete: In computer graphics, there is for instance the well-known, important distinction between manifold and non-manifold triangle meshes (the latter creating notorious problems).

Meshes are by no means the only way to describe shape. Creative minds have developed a great variety of different shape representations in the past decades. They all differ in their strengths and weaknesses, and it is often far from trivial to convert from one representation to the other. The mathematical theory of shape, however, applies to all of them. So it is quite important to be conscious about the theoretical background. Many shape representations developed by practitioners are surprisingly close to the abstract theory, point clouds being a notable example.

New ideas can also come from theory. One such idea, and a central idea pursued in this thesis, is to understand shape as the result of applying a sequence of operations, rather than just a list of geometric primitives. A list of triangles can represent a shape, but a sequence of Euler operators shows how to build the shape. So Euler operators are an example for the paradigm shift from objects to operations, for meshes as one specific shape domain. Meshes are only one example though, which should be kept in mind when reading this chapter. The same approach is certainly valid for other shape representations as well, since all methods are based on the same underlying theory, and they all must respect the same *nature of shape*.

The material presented in this chapter has been collected and combined, sometimes with some slight variations, mainly from two fabulous books. The book *Topology of Surfaces* from L. Christine Kinsey [Kin93] is a great introduction to the mathematical theory of cell complexes, surfaces, and manifolds, and the book *Introduction to Solid Modeling* from Martti Mäntylä [Män88] is the central resource for Euler operators and the conceptual bridge to shape modeling and CSG.

2.1 Basic Facts of Algebraic Topology

The objective of topology is the classification and description of the shape of a space up to topological equivalence. Most of the shapes computer graphics and geometric modeling are concerned with are of course embedded in Euclidean space.

Many ideas and algorithms on shape developed in these fields actually understand a shape just as a compact set of points in three-space. The familiar Euclidean space already has very much structure, but algebraic topology offers ways to reduce this structure to its very essence.

Definition 2.1 (Topological Space)

A topological space is a set X with a collection \mathcal{B} of subsets $N \subset X$, called neighbourhoods, such that

- for every point $x \in X$ there is a neighbourhood $N \in \mathcal{B}$ that contains x , and
- the intersection $N = N_1 \cap N_2$ of any two neighbourhoods $N_1, N_2 \in \mathcal{B}$ is again a neighbourhood $N \in \mathcal{B}$.

The set \mathcal{B} of all neighbourhoods is called a basis for the topology on X .

Definition 2.2 (Open sets, Topology)

Let X be a topological space with basis \mathcal{B} . A subset $A \subseteq X$ is an open set if each point $x \in A$ has a neighbourhood $N \in \mathcal{B}$ that contains x , and N is contained in A : $x \in N \subseteq A$. The set of all open sets \mathcal{T} is a **topology** on the set X .

Topology is an amazingly general concept, which can be illustrated by the example of a *discrete topology*: Let $X = \{x, y\}$. Then possible topologies \mathcal{T} on X include $\{X, \emptyset\}$, $\{X, \emptyset, \{x\}\}$, $\{X, \emptyset, \{y\}\}$, and $\{X, \emptyset, \{x\}, \{y\}\}$. But topology is also very fundamental and elegant, since the familiar definitions of interior, limit point, closed sets, connectedness and continuity can all be rewritten to depend only on the ideas of neighbourhoods and open sets. Note that X is required to be only a set, so no structure at all is needed to construct a topological space – just a collection of neighbourhoods that covers X . And the subsets obtained from combining these neighbourhoods are just the open sets of X . The open sets are in fact defined by the neighbourhoods, as a set is open if and only if it can be written as a union of the elements of the basis \mathcal{B} .

Theorem 2.3 (Elements of a topology)

Let X be a topological space with topology \mathcal{T} and basis \mathcal{B} .

1. X and \emptyset are elements of \mathcal{T} .
2. The union of any collection of elements in \mathcal{T} is in \mathcal{T} .
3. The intersection of any finite collection of elements in \mathcal{T} is in \mathcal{T} .

Definition 2.4 (Closed sets)

If C is a subset of a topological space X with topology \mathcal{T} , then C is closed if $X - C$ is open.

Theorem 2.5 (Closed sets)

Let X be a topological space.

- X and \emptyset are closed
- The intersection of any collection of closed sets is closed.
- The union of any finite collection of closed sets is closed.

So the closed sets are only derived from the open sets, defined as their set-theoretic complement. In 3D, the closed sets match the intuitive notion of a solid: It is natural to assume that the two-dimensional surface of a 3D solid belongs to the shape itself. This fact is important, e.g., for set-theoretic operations such as union, intersection and difference, which are actually used in practice with the CSG method to create 3D shapes. One subtle problem with this method is that the difference $A - B$ of two overlapping solids A and B , both closed sets, also removes B 's surface, and it leaves $A - B$ a set that is neither open nor closed. Fortunately, this issue can be resolved by a slightly modified, ‘regularized’ version of the operation: Instead of $A - B$, it yields the closure of the interior of $A - B$. Please consult Mäntyläs book for some interesting details [Män88].

The discrete topology mentioned before is just one example for topological spaces with seemingly counter-intuitive properties. In fact quite a bit can go wrong at the lowest level, and it is very interesting to see which low-level properties are needed to assure that things are as one would usually expect them to be. ‘Point set topology’ offers some definitions that clarify on such elementary properties:

Definition 2.6 (Hausdorff space)

A topological space X is a Hausdorff space if for every pair of distinct points $x, y \in X$, there are disjoint open sets U and V so that $x \in U$ and $y \in V$.

This definition is related to a classification of how powerful a topology is in separating different points and sets, namely the separation axioms. They define a hierarchy of T_0 , T_1 , and T_2 , or Hausdorff, spaces. And even stronger than Hausdorff spaces are ‘normal’ spaces, where every pair of disjoint closed sets is contained in a pair of disjoint open sets – which is a quite intuitive precondition, but by no means true for all topological spaces. Fortunately, there are easy ways to construct also spaces of a more familiar type.

Definition 2.7 (Metric, Metric Space, Metric Topology)

Let X be a set with a function $d : X \times X \rightarrow \mathbb{R}$. X is a metric space when d is a metric, i.e., when it satisfies

- $d(x, y) = 0$ if and only if $x = y$
- $d(x, y) = d(y, x)$ for all $x, y \in X$ (symmetry)
- $d(x, z) \leq d(x, y) + d(y, z)$ for all $x, y, z \in X$ (triangle inequality)

The metric d can be used to derive a topology on X , called the metric topology. It is defined by a neighbourhood which is the collection of all sets of the form $N = D_X(x, r) = \{y \in X : d(x, y) < r\}$ with $x \in X$ and a real number $r > 0$.

Definition 2.8 (Topology on Euclidean Space)

Points in the n -dimensional Euclidean space \mathbb{R}^n are denoted using coordinate tuples $\mathbf{x} = (x_1, x_2, \dots, x_n)$.

The Euclidean distance between two points $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ is given by the metric

$$\|\mathbf{x} - \mathbf{y}\| = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots + (x_n - y_n)^2}.$$

The open ball with radius r centered at $\mathbf{x} \in \mathbb{R}^n$ is $D^n(\mathbf{x}, r) = \{y \in \mathbb{R}^n : \|\mathbf{x} - \mathbf{y}\| < r\}$.

The open balls form a basis for a topology on \mathbb{R}^n .

So the familiar standard topology on \mathbb{R}^n is just a direct result of the standard Euclidean distance used as a metric. This definition works for any dimension. For $n = 1$ the induced metric topology is just formed by the open intervals. The Cartesian product of open intervals forms open rectangles in 2D and open boxes in 3D. Remarkably, it makes no difference if balls or boxes are used, since the resulting topologies are equivalent: An infinite union of balls can form a box, and an infinite union of boxes can form a ball. So both sets of neighbourhoods in fact agree on which sets are open. And both variants are actually used as shape representations in computer graphics! They are called ‘union of balls’ [ACK01a, ACK01b] and ‘octree’ or ‘voxel’ [CDM*02] representations, respectively.

Definition 2.9 (Continuity, Invertibility of Functions)

Let $D \subseteq \mathbb{R}^n$ and $R \subseteq \mathbb{R}^m$. A function $f : D \rightarrow R$ is continuous if whenever \mathbf{B} is an open set in R , then $A = f^{-1}(\mathbf{B})$ is an open set in D . The function is invertible if there is another function $g : R \rightarrow D$ so that $1_R = f \circ g : R \rightarrow R$ is the identity function of R , and $1_D = g \circ f : D \rightarrow D$ is the identity function of D . Then g is called the inverse f^{-1} of f .

Definition 2.10 (Homeomorphism)

A function $f : D \rightarrow R$ is a homeomorphism if it is both continuous and invertible, and the inverse function f^{-1} is also continuous. The spaces D and R are then topologically equivalent. Topological equivalence is an equivalence relation as it is reflexive, symmetric and transitive. A quantity is called a **topological invariant** if it is the same for topologically equivalent spaces.

Homeomorphisms are the ‘silver bullet’ of topology, since they allow to identify different topological spaces if they have basically the same structure: All that is required is that the open sets on both spaces are ‘compatible’. Seemingly quite different spaces can be mapped to each other, and are thus the *same* from a topologist’s point of view. An example are the infinite two-dimensional plane and the sphere, but only with the north pole missing, called the ‘pointed’ sphere. Both spaces are topologically equivalent since there is a homeomorphism, the stereographic projection: Put the sphere on the plane such that the south pole just touches the plane. Then any line from the north pole to a point on the plane touches the sphere at a unique point, and this establishes a mapping that is both continuous and invertible.

Continuity is of course much weaker than differentiability: The analytical properties of the spaces to be identified do not matter. So homeomorphisms are allowed to stretch and bend quite freely, but the spaces have to remain connected: A topological space is connected if it can not be written as a union of two (non-empty) disjoint open sets. Connectedness is a topological, or *intrinsic* property, whereas for instance boundedness is not: The sphere is bounded, but not is the plane. Consequently, as Kinsey says “no topological property should be based solely on distance, since in topology distance means very little” ([Kin93], section 2.4, p. 29).

A little stronger than boundedness is compactness, and it turns out to be a topological property: A set A is (sequentially) compact if every infinite sequence of points in A also has a limit point in A . With the standard topology on \mathbb{R}^n , the Heine-Borel theorem states that this is equivalent to saying that A is closed and bounded. So together with closedness, boundedness does qualify for being a very basic topological property. A subtlety though is that this is the case only for topologies that are not discrete. Algebraic topology has developed a powerful machinery and many different ways to tie spaces together, and to construct useful mappings between them. Only the most basic ones are presented next, and they directly relate to the construction of polygonal meshes.

Note by the way the difference between a homeomorphism and a (group) homomorphism: The latter is just a function f that is compatible with some group operations \star, \star' so that $f(x \star y) = f(x) \star' f(y)$. An invertible homeomorphism is

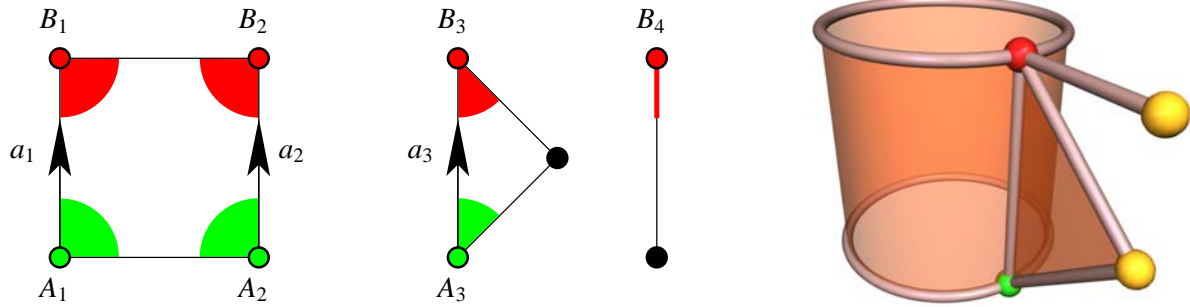


Figure 2.1: Example of an identification space. First a cylinder is constructed from a quadrangle by gluing two edges together: Edges a_1 and a_2 are identified through an equivalence relation (see Def. 2.13). In particular, the top and bottom pairs of vertices are identified. The neighbourhoods of these vertices are two half discs (with red and green halves) that extend over the left and right sides. Note that these half-discs are open in the relative topology. Another triangular 2-cell is attached by gluing a_3 to $a_1 \sim a_2$, and vertex B_4 of a 1-cell is identified with the top quad vertices. The resulting relations are $A_1 \sim A_2 \sim A_3$, $B_1 \sim B_2 \sim B_3 \sim B_4$, and $a_1 \sim a_2 \sim a_3$. The red vertex B is a complex vertex: Its neighbourhood is not homeomorphic to the open two-dimensional disc.

called isomorphism. Algebra and algebraic topology are both interested in functions that maintain some sort of qualitative structure. A quantitative study of analytic geometric properties on the other hand is the objective of differential geometry.

Definition 2.11 (Relative Neighbourhood, Subspace Topology)

Let $A \subset \mathbb{R}^n$. A neighbourhood of a point $\mathbf{x} \in A$ relative to A is a set of the form $D^n(\mathbf{x}, r) \cap A$.

More generally, let $A \subset X$ be part of a topological space X . Then the neighbourhood of a point $x \in A$ relative to A is a set $N \cap A$ where N is a neighbourhood of \mathbf{x} in X . The topology \mathcal{T}_A induced by this basis is called the subspace topology on A induced by the topology \mathcal{T} on X .

The subspace topology offers a way to obtain neighbourhoods of spaces (or objects) that are embedded in a larger space. In \mathbb{R}^n for instance, when $B \subseteq A \subseteq \mathbb{R}^n$, B is open or closed relative to A iff $B = A \cap C$ for some $C \subseteq \mathbb{R}^n$ that is open or closed, respectively. This way one can also speak of the neighbourhood of a point on a surface that is embedded in 3D, such as the surface of a solid. The induced 2D neighbourhood of a surface point \mathbf{x} is just the intersection of a 3D open ball $D^n(\mathbf{x}, r)$ around the point with the surface. The same works of course for one-dimensional curves embedded in 3D. The portion of the surface or the curve that is cut out of 3-space by the ball can have a very interesting structure. It is by no means in all cases just a plain local 2D or 1D space.

Another rich source of examples is the category of quotient or identification spaces. They are a device to filter out redundant information in contrast to the product spaces mentioned before which help to lift spaces to higher dimensions and thus add information.

Definition 2.12 (Quotient topology)

Let X be a topological space with topology \mathcal{T} and $f : X \rightarrow Y$ a function ‘onto’ another set Y , which means that the image of f covers Y completely. The quotient topology \mathcal{T}' on Y is derived from \mathcal{T} by defining U to be open in Y if $f^{-1}(U)$ is open in X . Thus, $U \in \mathcal{T}'$ if and only if $f^{-1}(U) \in \mathcal{T}$.

Definition 2.13 (Identification space)

Let X be a topological space with an equivalence relation \sim defined on X .

The equivalence class of $x \in X$ is $[x] = \{y \in X : x \sim y\}$.

The identification space X/\sim is defined as the set of equivalence classes of the relation \sim , so $X/\sim = \{[x] : x \in X\}$.

This operation is also called gluing.

An identification space is just a way of saying that x is glued to any y that satisfies $x \sim y$. Technically, the gluing operation is determined by the equivalence relation. A simple example is the construction of a cylinder X/\sim out of a unit rectangle X with an equivalence relation \sim like this: The equivalence class $[(x, y)]$ of a point (x, y) from the rectangle is a set $\{(x, y)\}$ containing only the point itself, unless $x = 0$ or $x = 1$, where it consists of the two equivalent points $\{(0, y), (1, y)\}$. Given a topological space X and an identification space X/\sim , there is a natural function $X \rightarrow X/\sim : x \mapsto [x]$ which simply maps an element to its equivalence class. Now this function induces a quotient topology on the identification space X/\sim . The point-wise equivalence therefore extends to whole neighbourhoods, just as one would expect from a real gluing operation.

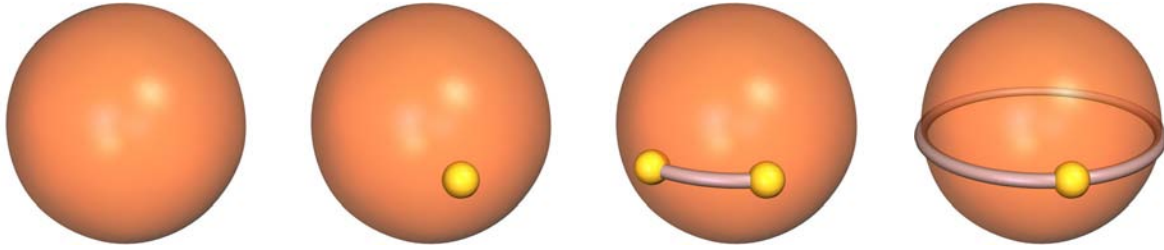


Figure 2.2: Cell complex on a sphere: The first is not a complex, because there is no lower-dimensional cell to bound the 2-cell. The insertion of a single 0-cell is sufficient to fulfil the condition (2) for cell complexes from Definition 2.15. The object (b) is the *pointed sphere*. The ‘sphere with a zipper’ (c) with one and the object (d) with two 2-cells are also often used as complexes on the sphere.

The cylinder example is elaborated in Fig. 2.1, where a topologically more complex object is created (left picture). The schematic representation to the left is called a *planar diagram*. The planar diagram has the advantage over the picture that all parts are visible and unambiguously denoted. And to a certain degree, the embedding shown in a picture is just an artifact: There are infinitely many possible embeddings. It is convenient to exclude embeddings that are too weird by imposing a few regularity conditions.

Definition 2.14 (n-Cell)

An *n-cell* is a set whose interior is homeomorphic to the *n-dimensional open unit disc* $D^n = \{\mathbf{x} \in \mathbb{R}^n : \|\mathbf{x}\| < 1\}$ with the additional property that its boundary must be divided into a finite number of lower-dimensional cells, called the *faces* of the *n-cell*. Notation: $\sigma < \tau$ if σ is a face of τ .

Definition 2.15 (Cell Complex)

A cell complex $K = \bigcup\{\sigma : \sigma \text{ is a cell}\}$ is a finite set of cells such that

1. if σ is a cell in K , then all faces of σ are elements of K
2. if σ and τ are cells in K , then $\text{Int}(\sigma) \cap \text{Int}(\tau) = \emptyset$

The dimension of the cell complex K is the dimension of its highest-dimensional cell. The set of all points in all cells in K is denoted $|K|$ and called the *realization* or the *support* of K . All *k-dimensional* (or short *k-cells*) of K form the *k-skeleton* of K .

Cell complexes are the basic device for expressing agglomerations of identification spaces in a *strictly* hierarchical fashion. They are frequently used as a theoretical device for arguing about shape, much like Turing machines serve as an abstract computer model. The basic requirement is only that cells are (relatively) open sets, bounded by lower-dimensional cells as their (relative) closure. 2-complexes are a very general form of polygonal meshes, although in practice meshes can occur that do not belong to the class of cell complexes. This is due to the fact that meshes may contain self-intersections, which is not allowed for cell complexes. This and a few other subtleties are the reason why the actual definition of meshes is postponed to section 2.3. Until then, the term ‘mesh’ will be used synonymously to ‘2-complex’. The following terminology is used for meshes in computer graphics:

- 0-cells, or point cells, are called *vertices*,
- 1-cells are line or curve segments, are called *edges*, and
- 2-cells are regarded as surface patches, are called *faces*.

Definition 2.16 (Degree, Valence)

The *vertex valence* is the number of times a vertex appears as an endpoint of an edge.

The *face degree* is the number of vertices, which equals the number of edges on the face boundary.

Especially relevant for computer graphics is the case where all 2-cells are triangles.

Definition 2.17 (Simplicial complex)

A *k-cell* is called a *k-simplex* if its boundary is formed by $k + 1$ $(k - 1)$ -cells. 1-simplices are line segments, 2-simplices are triangles, 3-simplices are tetrahedra. A cell complex is called a *simplicial complex* if all its cells are simplices.

Note that the general definition of a cell complex does not require that a 2-cell must be bounded by 1-cells: The surface of a ball is a sphere, but this is not a 2-cell because it is not homeomorphic to the open disc. Just like in the stereoscopic

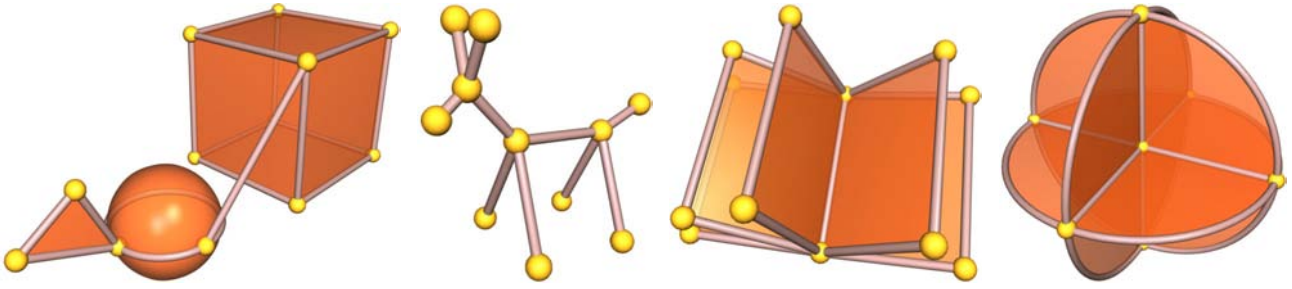


Figure 2.3: Different examples of cell complexes: (a) A solid cube with a hollow sphere and a triangle, attached by an edge and a point, (b) a set of connected line segments, (c) multiple sheets connected via a single edge, and (d) a vertex attached to six edges and twelve surfaces.

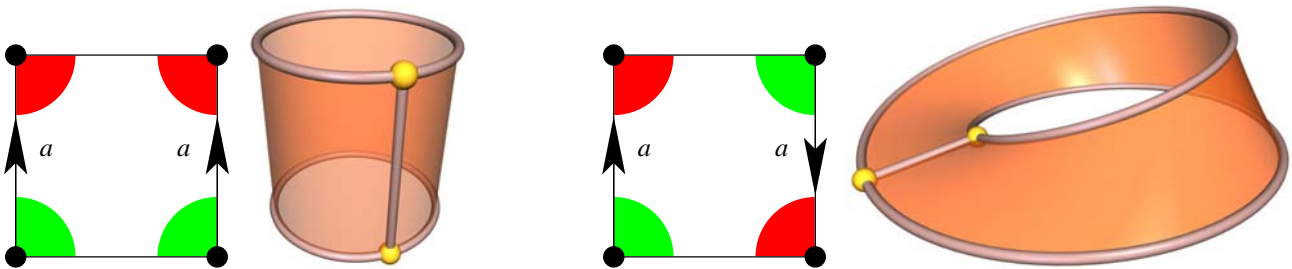


Figure 2.4: Cylinder vs. Möbius band. The planar diagrams (a) and (c) of both surfaces are very similar. The neighbourhoods of the start and end vertices of the arrow are shown in red and green. For the Möbius band the direction of one edge is reversed before gluing. A bug that crawls along the top edge of (c) from left to right finds itself suddenly in the bottom left corner of the planar diagram just when it has arrived in the top right.

projection example, it becomes a 2-cell if a point cell is inserted in the surface that serves as its boundary (see Fig. 2.2). Analogously, a 2D disc bounded by a circle is not yet a cell complex. The circle becomes a 1-cell only if there is at least one vertex to bound it. And cell complexes are also sensitive to their embedding: It may not be that two cells, for instance two line segments, intersect, if there is no 0-cell at the intersection point; condition 2 forbids all the special cases related to self-intersections.

A topological space is formed by defining neighbourhoods for all points in $|K|$. For 2-complexes, this is done by taking a collection of polygons and identifying or gluing edges and vertices together. A complete labeling as it was done in Fig. 2.1 can be tedious, especially for the vertices. Alternatively, just the edges are labeled, but provided with arrows to define directions. The direction of a single edge can make a great difference, as is demonstrated in Fig. 2.4 with the examples of the Möbius band and the cylinder. The Möbius band is a quite fundamental surface.

Definition 2.18 (Orientability)

A non-orientable surface is one which contains a Möbius band.

The Möbius band has the interesting property of having only one side and only one border, in contrast to the cylinder: There is no way to paint the two sides of the band in different colors. This also means that there is no way to distinguish an interior and an exterior. So a Möbius band cannot be part of the surface of a solid object: The surface of a (compact) solid partitions space into two disjoint volumetric sets, its (finite) interior and its (infinite) exterior. This distinction is only global, but orientable surfaces also permit a local distinction between inside and outside of an object through a consistent orientation of the 2-cells. A 2-cell is oriented by assigning a rotation direction. This determines a traveling direction along the cell boundary: Each boundary cell is traversed in the direction of the rotation, and this also determines a cyclic order of the boundary cells.

The orientation also permits to distinguish between the two sides of a surface. By convention, the side where the rotation direction is counterclockwise (*CCW*) is outside, i.e., directed towards the exterior, and the clockwise (*CW*) side is directed towards the interior. A pair of neighbouring 2-cells is oriented consistently if their choice of inside and outside is the same. A consistent orientation is not possible for surfaces that contain a Möbius band, since the part where the Möbius band is does not permit a consistent orientation (try with the complex in Fig. 2.4 (d)).

To a great part, computer graphics uses an abstraction: Instead of representing a three-dimensional solid volumetrically, as a collection of 3-cells, the surface of the object is taken as the object itself. The abstraction lies in the fact that

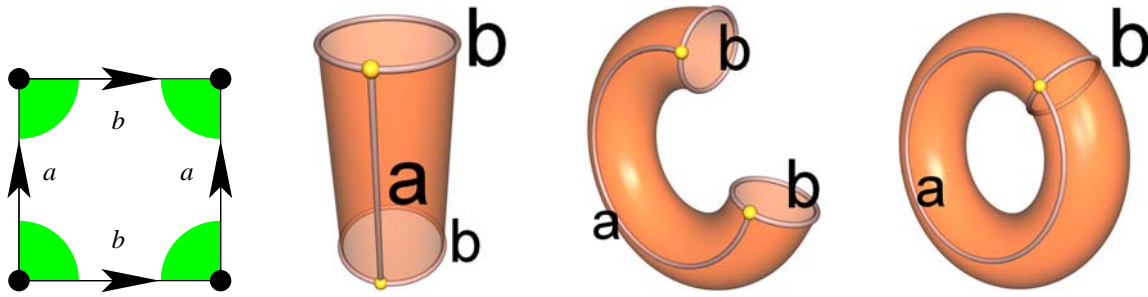


Figure 2.5: Construction of a torus via identification spaces. First edges a are identified to create a cylinder, as in Fig. 2.4. Then the cylinder is bent, and the top and bottom boundaries are also identified. The result is a closed surface. All four corners of the planar diagram are glued together, and its neighbourhood is a full 2-disc (green).

every real surface, such as a sheet of paper or metal, are of course not infinitely thin, but they are solid objects themselves. The surface of real-world objects though *is* closed as well as orientable. The objective of computer graphics is image synthesis. The image we get from a real world object is, unless it is translucent, determined by its surface, which justifies the abstraction mentioned. This makes surfaces a distinct object of study for computer graphics. The central notion for the study of surfaces is the concept of a manifold.

Definition 2.19 (Manifold)

An n -dimensional manifold is a topological space such that every point \mathbf{x} has a neighbourhood topologically equivalent to the n -dimensional open disc $D(\mathbf{x}, r)$ with center \mathbf{x} and radius r . Further is required that any two distinct points have disjoint neighbourhoods (Hausdorff). A 2-manifold is also called a surface.

An n -manifold with border is a topological space such that every point has a neighbourhood topologically equivalent to either D^n or the half-disc $D_+^N = \{\mathbf{x} = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n : \|\mathbf{x}\| < r \text{ and } x_n \geq 0\}$ (manifold with border).

Manifold surfaces are much more restrictive than 2-complexes: A 2-complex K is called a *manifold complex* only if its realization $|K|$ has the manifold property, i.e., every point of $|K| \subset \mathbb{R}^3$ has a neighbourhood that is homeomorphic to the 2-dimensional open (half-)disc. This is not the case for any of the cell complexes in Fig. 2.3 since they all contain whole edges that violate the manifold property.

Although only a subset of all 2-complexes can be embedded on manifold surfaces, it is nevertheless a very important subset. It also plays an distinct rôle in computer graphics, forming an important class of meshes. An even more restricted class of surfaces is also a fundamental, long-standing subject of study, namely surfaces that are exclusively composed of linear parts.

Definition 2.20 (Polyhedron)

A 2-complex K is a polyhedron if its support $|K|$ is a closed orientable manifold surface, its edges are straight line segments, and its faces are simple planar polygons.

Manifolds are also more general than cell complexes: They do not require any partition into cells or other kinds of sub-structures. They merely provide the support, or realization, $|K|$ for many cell complexes. So when speaking of a cell complex as a surface, the realization is meant. Different cell complexes can have the same support, and thus may define the same topological spaces. The five Platonic polyhedra, shown in Fig. 2.12 later, are all complexes on the sphere, an orientable 2-manifold. They can also be considered complexes on the solid ball, if the interior is counted as a 3-cell.

The question arises of how manifold surfaces can be categorized. The objective of algebraic topology is to classify topological spaces up to homomorphisms, i.e., up to topological equivalence. The 2-manifolds are a very restricted class of topological spaces, so there should be some hope to find a good classification. It can be obtained by using a very basic technique.

Definition 2.21 (Connected Sum of Surfaces)

Let S_1 and S_2 be two surfaces. Remove a small disc from each of them, and glue the boundary circles of these discs together. The new surface is called the connected sum of S_1 and S_2 , written as $S_1 \# S_2$.

The connected sum operation is a very powerful device for creating more complex shapes by gluing simpler surfaces together. One example, the connected sum of two tori, is shown in Fig. 2.7. This way, one can make up any combination of closed orientable and non-orientable surfaces, such as spheres, tori, and Klein bottles. There is also a ‘minimal’ closed non-orientable surface. It leads to a quite concise categorization of closed surfaces.

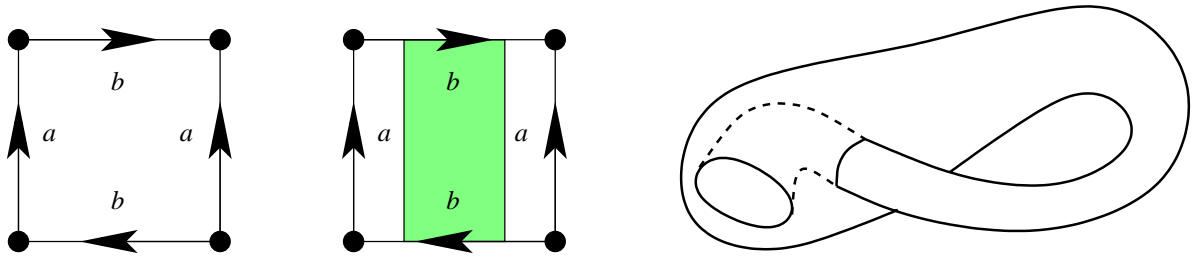


Figure 2.6: Klein bottle vs. Torus. Just as with the cylinder and the Moebius band, a Klein bottle is very similar to the torus, except that the orientation of one arc is reversed. The resulting surface is non-orientable, as it contains a Möbius band (green). But it is also a closed surface, so it cannot be embedded in 3-space without self-intersection, as sketched in (c).

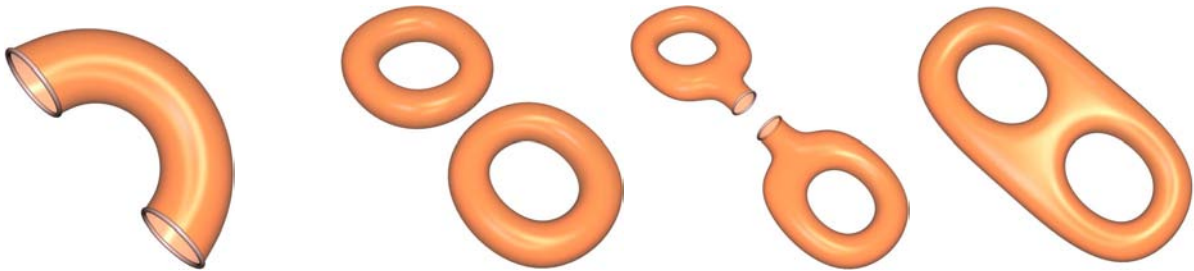


Figure 2.7: Connected sum of two tori. Orientable surfaces are classified as spheres with a number n of handles, such as the one in (a). This is topologically equivalent to the connected sum of n tori: A double torus is just a sphere with two handles. Image (c) shows two tori with open discs removed, so that the boundary circles can be glued together to form the double torus (d).

Definition 2.22 (Projective plane)

The projective plane \mathbb{P}^2 is the connected sum of the sphere \mathbb{S}^2 and the Möbius band. So it is constructed by cutting out a disc from the sphere and identifying the boundary created with the boundary of the Möbius band.

Surprisingly, a surface with border can be handled very much like a closed surface. The reason is that the border can be naturally divided into a number of loops: It must form one or more closed cycles which are made up from the (overlapping) neighbour half-discs of the border points. The surface can be closed when these border loops are filled, by inserting additional 2-cells, each of them homeomorphic to the open 2-disc.

Theorem 2.23 (Classification of surfaces)

The connected sum of a torus and a projective plane is topologically equivalent to the connected sum of three projective planes: $\mathbb{T}^2 \# \mathbb{P}^2 = \mathbb{P}^2 \# \mathbb{P}^2 \# \mathbb{P}^2$. This leads to the following classification of compact connected surfaces:

- A closed surface is homeomorphic to a sphere, a connected sum of n tori, or a connected sum of n projective planes.
- A closed orientable surface is therefore homeomorphic to either a sphere or a connected sum of n tori.
- The same applies to a surface with borders, except that a finite number of discs is removed.

The proof of this quite important theorem is constructive. First, it can be shown that every compact connected surface admits a triangulation. This is a simplicial complex with only finitely many triangles. It can also be shown that a surface is connected if and only if the triangles can always be arranged in a list such that triangle T_i can be glued to a triangle T_j , $j < i$ earlier in the list. But then there must also be a connected planar diagram of all the triangles, this is basically a set of very long strips of triangles forming a tree. The interior edges are irrelevant, but all edges on the border have labels to still be matched. Now the proof proceeds by performing the actual gluing, i.e., by shortening the border, until only the few prototype cases above remain. For details on this case distinction, see Kinsey [Kin93], section 4.4. She also presents a simple 15 step reduction from $\mathbb{T}^2 \# \mathbb{P}^2$ to $\mathbb{P}^2 \# \mathbb{P}^2 \# \mathbb{P}^2$ to prove the key fact that a combination of tori and projective planes can be reduced to a combination of projective planes alone.

In the domain of closed 2-manifolds, the main distinction is therefore between orientable and non-orientable surfaces. Non-orientable surfaces can be considered pathological cases for the purposes of computer graphics, as there is almost no practical use for them. Fortunately, it is not very difficult to determine whether a given 2-complex is a closed, orientable

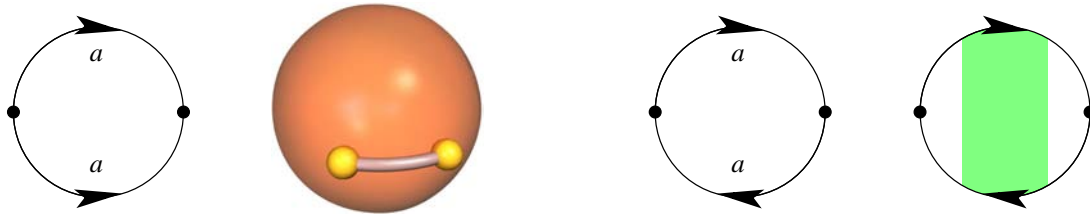


Figure 2.8: Planar diagrams of sphere (a), (b) and crosscap (c), (d). The projective plane \mathbb{P}^2 , also called crosscap, is similar to the ‘sphere with a zipper’, with the difference that the orientation of one arc is reversed. The result is that a Möbius band (green) is part of the surface (d), which makes the surface non-orientable. It can be imagined as one side of the zipper on the sphere being reversed, a similar relation as between the cylinder and the Möbius band (Fig. 2.4).

surface	$ V $	$ E $	$ F $	genus	χ
sphere	1	0	1	0	2
n tori	1	$2n$	1	n	$2 - 2n$ negative, even
m projective planes	1	m	1	m	$2 - m$ odd, negative for $m > 2$

Figure 2.9: Genus and Euler characteristic of the typical prototype surfaces.

surface: A consistent orientation of the 2-cells must be possible, and the relative neighbourhood of every point on the surface must be homeomorphic to the open 2-disc. The first property, orientability, can be checked locally in a similar way as in the proof sketch above. The second property is trivial for 2-cells, and has to be checked only for 1- and 0-cells. In the terminology of computer graphics, this provides a criterion to determine the (topological) consistency of a mesh.

Theorem 2.24 (Mesh consistency)

A 2-complex is called a valid mesh if it is a closed, orientable manifold surface; otherwise it is a general or invalid mesh. A mesh is valid if and only if it fulfills the following criteria.

1. **Manifold Edge Property:** Every edge is incident to exactly 2 faces.
2. **Manifold Vertex Property:** Each vertex has an edge cycle of all incident edges. This is a cyclic list of edges such that each pair of consecutive edges in the list, together with the vertex itself, constitutes a part of the boundary of some face.
3. **Orientability:** All faces can be oriented in a consistent fashion. This means for all edges that the faces on both sides traverse this edge in opposite directions, according to their respective orientations.

This criterion allows to check in a straightforward manner whether a given 2-complex, or mesh, K is closed, manifold, and orientable. It is also possible to derive the type of surface that permits to embed K without self-intersections, according to the classification from theorem 2.23. This is possible with the Euler characteristic of K .

Definition 2.25 (Euler characteristic, Genus)

Let K be a complex. The Euler characteristic of K is $\chi(K) = |0\text{-cells}| - |1\text{-cells}| + |2\text{-cells}| - |3\text{-cells}| \pm \dots$

The Euler characteristic of a 2-complex is the alternating sum $\chi(K) = |V| - |E| + |F|$.

The genus of a compact surface S is $\frac{1}{2}(2 - \chi)$ if S is orientable, and $2 - \chi$ if S is non-orientable. Correspondingly the numerus of a mesh is the number of its shells: A shell is a connected components in a mesh or 2-complex.

The genus is more related to physical properties of the surface, such as the number of handles or the number of crosscaps, that are attached to the sphere. The sphere is the prototype genus 0 object. The Euler characteristic on the other hand is 2 for the sphere, and this is its maximum value. Neither Euler characteristic nor genus can reliably distinguish between orientable and non-orientable surfaces: Not all surfaces with the same Euler characteristic are topologically equivalent.

Theorem 2.26 (Euler characteristic)

Two compact connected surfaces are topologically equivalent if and only if they have the same Euler characteristic, they are both either orientable or non-orientable, and they have the same number of borders.

Any 2-complex K such that $|K|$ is topologically equivalent to the sphere has Euler characteristic $\chi(K) = 2$.

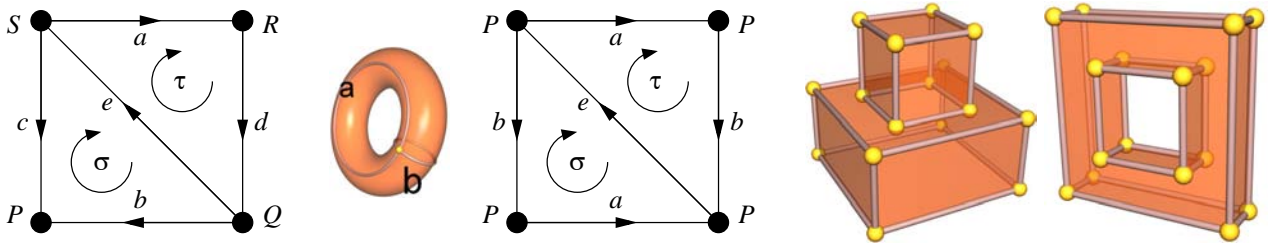


Figure 2.10: Directed Complex and Ring Examples. A directed complex on a surface with border (a), and a complex on the torus (b), (c). The connected boxes (d) and the quad torus (e) contain faces with rings. They are therefore no cell complexes in the strict sense.

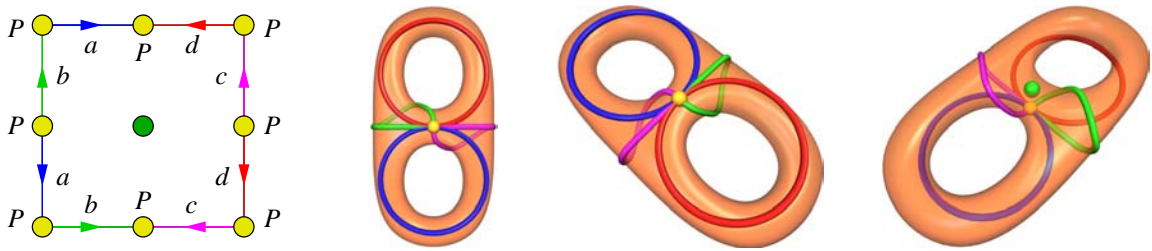


Figure 2.11: Four closed paths on a double torus. The Betti number β_1 of the double torus is 4: The surface is still connected, as any point on the surface can be reached from the green point without crossing any of the paths. Note that all eight yellow vertices on the boundary of the planar diagram (a) are identified and fall onto the same point in the embedding.

This means that the Euler characteristic is *not* a topological invariant, but almost: For instance on the set of all orientable 2-complexes without borders, it is. – The proof of the theorem is again constructive; it uses a reduction where edges, faces and vertices are removed until no further removal is possible. Each removal leaves the Euler characteristic constant, and thus all that remains to be proven is the Euler characteristic for the prototype surfaces, the sphere and the n -handled torus. The respective numbers of vertices, edges, and faces are listed in table 2.9. Note that the minimal genus-0 object, the pointed sphere shown in Fig. 2.2 (b), consists of one face and one vertex only. Certain objects are also described more easily when allowing *rings*, as in the last two images Fig. 2.10 (d) and (e): A face with rings has two or more disjoint boundary polygons, so it is not homeomorphic to the open disc. Yet it is always possible to insert an edge that connects a ring with the outer boundary polygon, thus re-establishing the open disc property. Note that this edge is incident to the same face on both sides. The remarkable thing about the Euler characteristic is that it can be related to the number of connected components and the total number of handles in all these connected components.

Theorem 2.27 (Extended Euler-Poincaré formula)

Let K be a complex on a compact orientable surface. Let $v = |V|$, $e = |E|$, $f = |F|$ be the number of vertices, edges, and faces of K . Then

$$v - e + f = 2(s - h) - b + r ,$$

where s is the number of connected components, also called shells, h is the number of topological holes, b is the number of border components, and r is the number of rings.

The basic form of the extended Euler-Poincaré equation is $v - e + f = 2(s - h)$. This equation, however, is far from obvious. It deserves some explanation. The Euler-Poincaré formula is a specialization for 2-complexes of a more general equation from a theorem for n -complexes which states that $\chi = \beta_0 - \beta_1 + \beta_2 - + \dots + (-1)^n \beta_n$. It is proven in Kinsey’s book [Kin93] as theorem 6.24. The β_i are the *Betti numbers*. They are a result of *homology* theory, another subject in algebraic topology, which is a group-theoretic approach to a better surface classification. As was noted above, the Euler characteristic cannot distinguish between orientable and non-orientable surfaces, which is a quite elementary distinction. This drawback can be remedied when providing each cell of an n -complex with an orientation, which can be understood as a traveling direction on the cell. Longer paths, called k -chains, can be formed by summing up such k -cells, which yields the *algebra of chains*. The interesting thing now is the boundary operator ∂ . The boundary of a k -cell σ is the chain of $(k - 1)$ -cells that are faces of σ , but with the orientation inherited by σ . In the example in Fig. 2.10 (a), $\partial(\sigma) = b - c - e$,

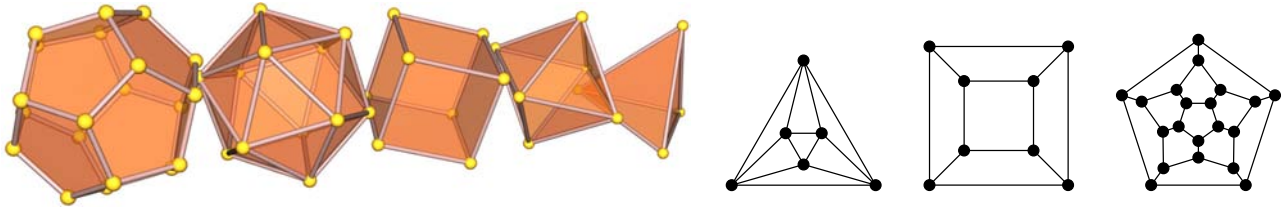


Figure 2.12: The five Platonic solids: The dodecahedron and icosahedron are dual, cube and octahedron are dual, and the tetrahedron is self-dual. The plane models of octahedron, cube, and dodecahedron are created by stretching one face. This face turns into the unbounded exterior.

and $\partial(b) = P - Q$. Thus, the boundary operator maps a k -chain to a $(k - 1)$ -chain:

$$\partial(a_1\sigma_1 + \dots + a_m\sigma_m) = a_1\partial(\sigma_1) + \dots + a_m\partial(\sigma_m)$$

In example (a), which is a surface with border, $\partial(\sigma + \tau) = a + b - c + d$, as e is canceled away because it appears in both directions in σ and τ . A slight variation yields (b), the familiar complex on a torus, now with a diagonal. The torus is a closed surface, and here $\partial(\sigma + \tau) = 0$. In particular, any multiple $i\sigma + i\tau$ is also mapped to 0 by the boundary operator. The kernel of the boundary operator, the set of k -chains that are mapped to 0, is called the k -cycles. For any complex on the torus, the set of 2-cycles, like the group $\{i(\sigma + \tau), i \in \mathbb{Z}\}$ from image (b), is isomorphic to \mathbb{Z} . And this is exactly what is measured by the second Betti number β_2 , which is also equal to the number of cavities enclosed by a complex. The Klein bottle cannot enclose a cavity, i.e., it does not partition \mathbb{R}^3 in two distinct sets, an interior and an exterior. Non-orientable surfaces have $\ker(\partial, 2\text{-chains}) \simeq \{0\}$ and thus $\beta_2 = 0$.

The idea of homology is to measure the power of the chain groups: Two k -chains are homologous if their difference is the boundary of some $k + 1$ -chain. Two vertices, for instance, are homologous if there is a path between them, and β_0 is the number of equivalence classes of vertices with respect to homology. But this is just the number of connected components, which equals the number of cavities, if all components are orientable. So for orientable surfaces with multiple connected components, or multiple *shells*, $\beta_0 = \beta_2$. Betti number β_1 can also be interpreted as the number of closed paths that can be put on a surface without decomposing it into disjoint parts. This number is 0 for the sphere, as any closed path on the sphere, such as the equator, creates two disjoint surface parts. Yet two such paths can be put on the torus, as for instance a and b in image 2.10 (b), without affecting the connectedness of the surface. And on the double torus, even four closed paths can be inserted without separating any two surface points, as shown in Fig. 2.11. This demonstrates the fact that β_1 can be interpreted as twice the number of topological holes, or handles, of the surface. Finally, this yields the Euler-Poincaré equation as follows:

$$v - e + f = \chi = \beta_0 - \beta_1 + \beta_2 = s - 2h + s = 2(s - h)$$

The extended form of this equation as in the theorem is obtained by counting one additional edge for each ring, because 2-cells may only have a single boundary curve. The treatment of border components is just as in Theorem 2.23, where it was argued that each border loop corresponds to a face. So the left-hand side of the equation above actually reads $v - (e + r) + (f + b)$, which then yields the extended Euler-Poincaré equation.

The remarkable thing about this formula is that it allows to derive the type of surface which permits an embedding of K without self-intersections, by only looking at the numbers of vertices, faces, and edges. For a compact, connected, orientable 2-complex that has no rings, the number of topological holes is $h = \frac{1}{2}(e - v - f) + 1$.

2.2 Euler Operators

The considerations so far were based on the assumption that it is possible to manipulate surfaces and cell complexes, but there was no dedicated discussion of the devices that permit to do so. The surface classification theorem 2.23 for instance used a method to reduce a given complex to one of the standard complexes, which are the sphere and the n -handled torus for closed, orientable surfaces. Consequently, for creating new objects the point of departure are complexes on the sphere.

A very useful change of perspective reveals what is necessary to create genus-0 objects: To consider a 2-complex as a planar graph. Physically, this can be understood as taking one face of the complex, which is deemed to consist of a rubber-like material, and to stretch it so much that the rest of the complex is suspended in its interior. This way, so-called *plane models* are constructed, an embedding of a 2-complex in the 2-dimensional plane. Note the difference to the planar diagrams from the previous section: The unbounded exterior of the plane model actually belongs to it, as it corresponds

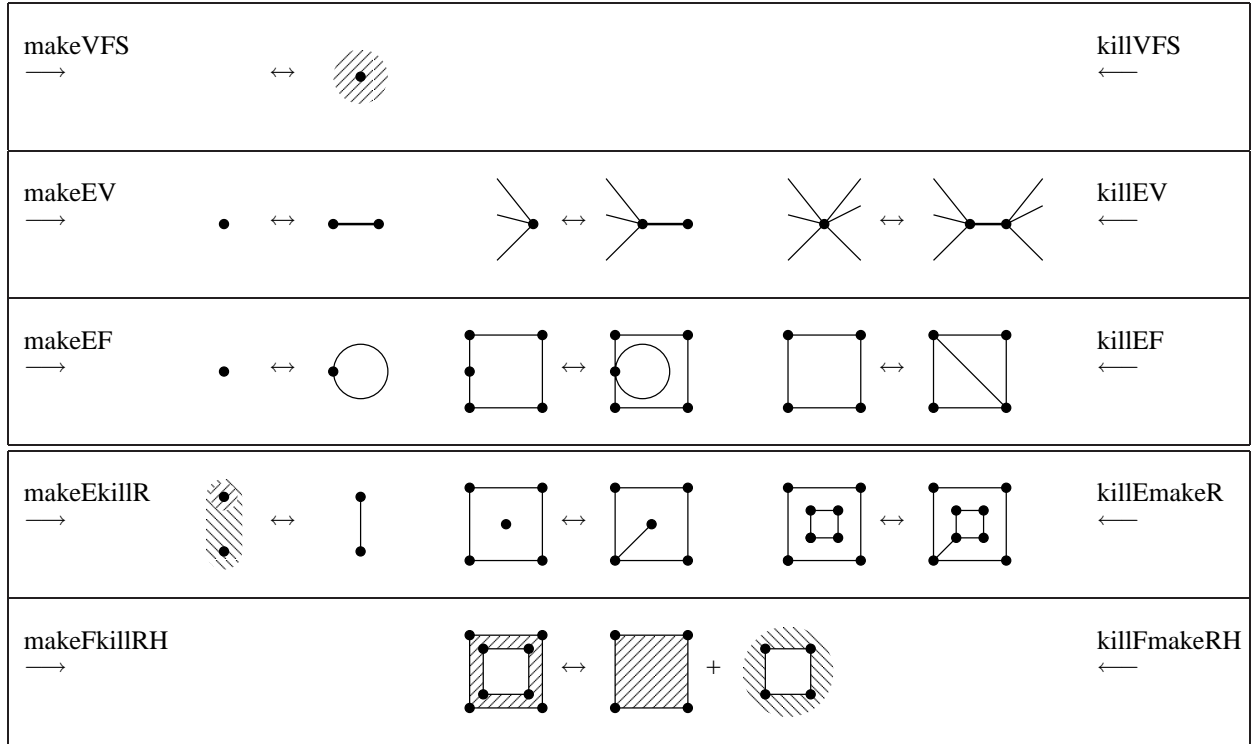


Figure 2.13: Planar models of Euler operators to manipulate a 2-complex. The make... operators in the left column correspond to their inverse kill... operators to the right: Each row shows different situations for applying the operator. The ↔ arrows can be either read from left to right (make...), or from right to left (kill...). With the first three operators, any genus 0 shape can be built. The last two operators are concerned with rings and building higher-genus objects. The different versions of the same operator are denoted makeEV (a), (b), (c), etc.

to the face that has initially been stretched. Note that the unbounded outer face is CW oriented, while the bounded faces retain their CCW orientation when transformed to a plane model.

The obligatory examples are the plane models of the Platonic solids in Fig. 2.12. They are completely regular because all faces are congruent, all edges have the same length, all vertices have the same valence, and the angles in all corners are equal. The Platonic solids also have the interesting property that the tetrahedron is dual to itself, the octahedron is dual to the cube, and the dodecahedron is dual to the icosahedron. Duality in this case is defined in the topological sense by exchanging the roles of faces and vertices:

Definition 2.28 (Topological Dual of a 2-complex)

Let $K = (V, E, F)$ be a 2-complex on a closed orientable manifold surface. The topological dual $\bar{K} = (\bar{V}, \bar{E}, \bar{F})$ is another 2-complex with $|\bar{V}| = |F|$, $|\bar{E}| = |E|$, and $|\bar{F}| = |V|$. Each vertex in the primal complex corresponds to a face in the dual complex and vice versa. Two vertices $\bar{v}, \bar{w} \in \bar{V}$ of the dual are connected if and only if the corresponding faces $f_{\bar{v}}, f_{\bar{w}} \in F$ of the primal complex are neighbours.

Every closed edge cycle around a vertex in the primal complex corresponds to a closed face boundary in the dual complex. Consequently, the sequence of edges on a face boundary is called the edge cycle of the face.

This establishes a correspondence between edge cycles around vertices in the primal and face boundaries in the dual complex. The dual of the dual complex is again the primal complex. To practically obtain the topological dual the vertices can be placed in the face centroids. Topological duality is also a justification for plane models, since it carries over to the duality defined for planar graphs, when considering the plane model of a 2-complex.

But not all 2-complexes can be embedded in the 2-plane as a whole, of course. An embedding of any higher-genus object has self-intersections; only partial embeddings are possible, which are then to be glued together (see Def. 2.13). This paradigm offers the useful view of a 2-complex as a *locally planar graph*. It reduces the problem of creating 2-complexes in 3-space to the question which kind of operations permit to create such graphs in the plane. One possible answer are *Euler operators*. There are five Euler operators, and each of them is invertible, which makes for a set of ten operators altogether. The table in Fig. 2.13 shows the effects of all ten operators and the different possibilities to apply

makeVFS	none
killVFS	shell has no edges and single vertex
makeEV	none
killEV	edge has different vertices on both ends
makeEF	vertices belong to the same face boundary of the same face
killEF	edge has different faces on both sides
makeEkillR	vertices belong to different face boundaries of the same face
killEmakeR	edge has the same face on both sides, edge is not a loop
makeFkillRH	can only be applied to a ring
killFmakeRH	a face may not become ring of itself

Figure 2.14: Conditions for legal application of Euler Operators.

each operator. Another table in Fig. 2.14 lists the conditions under which each operator can be legally applied. Before showing a concrete example that demonstrates how to build a torus, the individual operators are introduced. The first three Euler operators are enough to create any object that is topologically equivalent to the sphere.

makeVFS: Make Vertex, Face, Shell creates the minimal 2-complex, the pointed sphere. It provides one connected component (a shell) that consists of just a face and a vertex attached to it, to initialize the modeling process. The planar model actually consists of the vertex alone, the face is the (unbounded) plane around it. To better distinguish between the point with the plane and the empty configuration (with no plane), the plane around the point is hatched in Fig. 2.13, (1a).

makeEV: Make Edge & Vertex is the vertex split operator. It allows to split any vertex into a pair of separate vertices connected by a new edge. Thus it introduces one edge and one vertex, which explains the name. Different configurations are possible: The new vertex can be attached to an isolated valence-0 vertex, for example created by makeVFS (2.13, (2a)). When attached to a higher-valence vertex, there are two possibilities, either a *dangling vertex* (2.13 (2b)) or the classical vertex split (2.13 (2c)). The inverse operator killEV is also called *edge collapse*.

makeEF: Make Edge & Face is the face split operator. It is dual to makeEV as it splits a given face in two, introducing a new face and a new edge. The three possible configurations are also the duals of the makeEV configurations: The loop to create a 1-gon (2.13 (3a)), the ‘dangling loop’ inside a face (2.13 (3b)), and the classical face split (2.13 (3c)). Its inverse killEF is also called the *face join* operation.

Any sphere-like object can now be created, proceeding in a breadth-first manner, starting from a single initial seed-face for each shell: The first double sided face is created by one makeVFS, a sequence of makeEV (a), and one makeEF (c). One side is then further expanded by issuing some makeEV (b) and (c), and for every desired face one makeEF (c). The next two operations are concerned with genus changes: To create topological holes or to glue different objects together so that they form a single connected component. This can be elegantly accomplished by using rings as the key concept.

makeEkillR: Make Edge, Kill Ring connects a ring with the outer face boundary. This creates just a single edge to decrease the number of face boundaries by 1. The newly inserted edge is incident to the same face on both sides, with 2.13, makeEkillR (c) as the typical case. The ring however may as well contain only a single vertex (b). The inverse killEmakeR operator is also applicable to a shell that consists of only a pair of connected vertices (a), for example created by the sequence makeVFS, makeEV (a). In this case, the result are two points with attached faces, one face being a ring of the other face. This is suggested by the different hatches.

makeFkillRH: Make Face, Kill Ring & Hole turns a ring into a face of its own. This is maybe the simplest, but also the most abstract Euler operator. Its inverse, killFmakeRH, is actually easier to understand, because it exactly corresponds to the *connected sum of surfaces* from definition 2.21: One face a of a shell A is turned into a ring a' of one face b of another shell B , thus creating the connected sum $A\#B$ of both shells. The elegant thing is that in case $A = B$, exactly the same operation can be used to create a (topological) hole, changing the genus of an object. The diagram in Fig. 2.13, (5a) attempts at illustrating this operation: Given a face (hatched) with a ring (white interior), the makeFkillRH operation just disconnects the ring from the face. The face loses its ring (big hatched quad), and the ring becomes a face in its own right, shown as the hatched exterior around the white quad. Note that this exterior face is CW oriented, which is also the case when it is used as a ring in the bit quad on the left side.

So `makeFkillRH` and `killFmakeRH` just convert back and forth between faces and rings. A very illustrative example are the stacked boxes in Fig. 2.10 (d): They are connected by making the bottom face of the small box a ring of the top face of the large box via `killFmakeRH`. This makes only sense, however, if the small box is really stacked on top of the larger box, so that the respective faces are coplanar. In this case, by the way, the name `killFmakeRH` is a misnomer, since no hole is created, but two shells are joined. The operation should therefore be called `killFSmakeR` for ‘Kill Face & Shell, Make Ring’ when used for joining shells. Yet since the same operation can be used in two different ways, either name can do, so one may as well stick to the name `killFmakeRH`. This operation is also used with the other ring example from Fig. 2.10, the quad torus (e). Its construction via Euler operators is demonstrated in the following section.

2.2.1 Euler Operator Example: Quadrangular Torus

A concrete example explains best how Euler operators work in practice. The example model is the quadrangular torus, a quadrangular box with a quadrangular hole. It is assembled in twelve steps, illustrated in Fig. 2.15. The same steps are shown both as planar diagrams (rows 1,2,3) and in perspective view (rows 4,5,6).

Steps 1–2. Modeling starts with the pointed sphere, created by `makeVFS` (step 1). This first vertex is split three times, to create a chain of four vertices (step 2). The first split is of type `makeEV` (a), the next two splits add dangling vertices with `makeEV` (b). Note that the single resulting face is a hexagon, which can well be seen in the planar diagram: It takes six steps to travel around the vertices, for instance in counter-clockwise direction. In the perspective view, the face (shown in wireframe) is shown as ‘suspended’ in the polygon formed by the three edges.

Step 3. This shows the value of planar models: Some configurations tend to get complicated when realized in 3D; in such cases, the corresponding planar diagram can help a lot. This is also true for the next step: Via `makeEF`, the polygon is closed, and a double-sided quad is obtained. In the planar diagram of step 3, two faces can be clearly identified: The brown quad and the (infinite) green plane. In 3D, they are just the co-planar front- and backsides of a double-sided quad.

Steps 4–5. Four `makeEV` (b) operations in the corners of the brown quad create four dangling nodes and raise the face degree to 12. The corresponding perspective image shows the non-planarity of this face, which is a result of the dangling vertices sticking out of the original face plane. The planarity is re-established when four new quads are formed by four times applying `makeEF` (step 5). The result so far is a quad box with the topology of a cube: It has eight corners, six faces and twelve faces. And note that 13 Euler operations were used so far, all except the first with an ‘E’ in their name, six of them with an ‘F’, and eight containing a ‘V’.

Steps 6–8. One dangling vertex (step 6) forms the basis for another quad, created the same way as the first one in the beginning (steps 7,8). The brown quad is embedded in the green face, but thanks to the edge in the corner, the green face is still homeomorphic to the 2-disc. It has degree 10, and the diagonal edge is incident to the same face on both sides.

Step 9. One application of `killEmakeR` removes the diagonal edge and makes the brown face a ring of the green face, lying entirely in its interior. Note that the green face is no longer homeomorphic to the 2-disc. So the resulting surface is no longer a cell complex in the strict sense (see definition 2.15). However, keeping track of which ring belongs to which face, it is always possible to introduce ‘artifact’ diagonal edges to connect the rings (inner boundaries) with the outer boundary, so that the face is homeomorphic to the 2-disc again. This simple operation can restore the cell property of the face.

Steps 10–11. Just as in steps 4–5 before, four `makeEV` followed by four `makeEF` are applied to the quad face, resulting in a displaced copy of the original face. This displacement operation is called an *extrusion*, and it can of course be applied to faces of any degree, not just to quads. So two quad extrusions are used: step 3 → step 5, and step 9 → step 11. Note that also in the general case, the side faces are always quads. The second extrusion is in negative direction (with respect to the face normal), which can of course only be seen in perspective view, since the planar models of positive and negative extrusions are identical. The second extrusion is made such that the extruded face in step 11 lies exactly in the same plane as the face in step 3.

Step 12. The last operation, `killFmakeRH`, just makes the extruded face from step 11 a ring of the bottom face of the box. This also creates a topological hole in the object, hence the name of the operation. It is hard to illustrate this operator, as it does not at all change the connectivity of the 2-complex. It merely just flags a face to become a ring, and hereby the face ‘dissolves’. For the sake of clarity, steps 11 and 12 are also shown from below in Fig. 2.16. The left image (a)

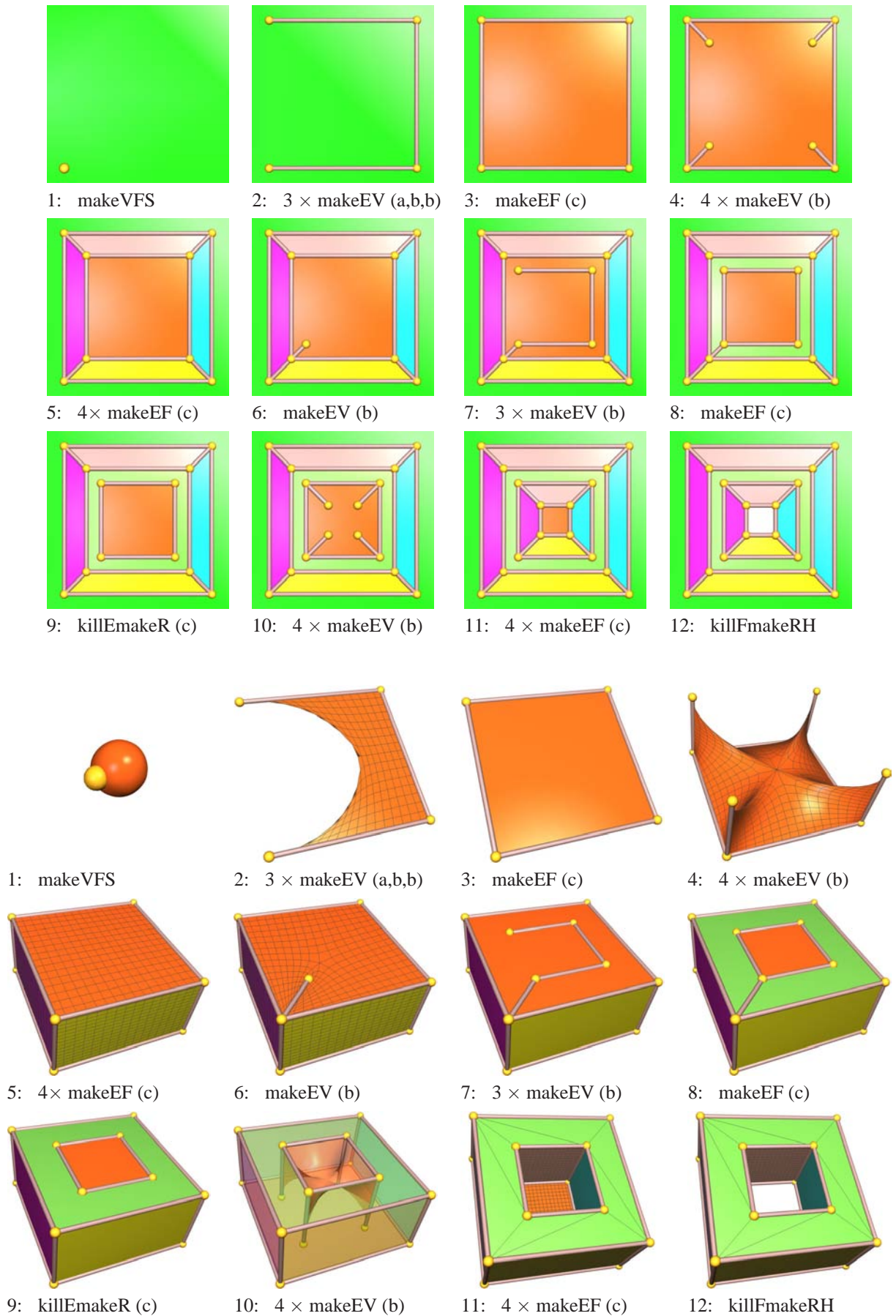


Figure 2.15: Euler operator example: Quadrangular Torus. For an explanation see section 2.2.1.

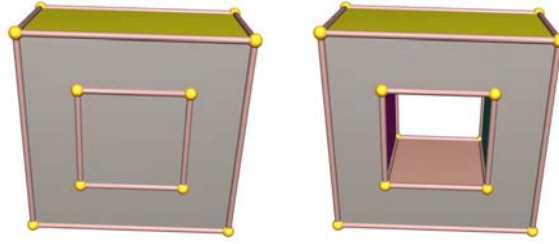


Figure 2.16: The creation of a topological hole using `killFmakeRH`. These are the last two steps 11 and 12 from the construction of the quad torus (Fig. 2.15), but shown from below. The left face has no rings but the vertices just ‘shine through’.

shows step 11: The extruded quad face (brown) cannot be seen, only its vertices and edges ‘shine through’. Note that the bottom face is oriented CCW, but the invisible extruded quad is oriented CW from the current viewpoint. The right image 2.16 (b) shows the situation after `killFmakeRH`. The extruded quad face and the inner part of the bottom face ‘cancel out’. What remains is a hole, and the grey face becomes incident to the former neighbours of the ‘dissolved’ face. Note that the ring retains the orientation of this face: Rings are always oriented clockwise. This is consistent with the rule that the face *interior is to the left* when traveling along the face boundary in the direction prescribed by the orientation.

2.2.2 Another way to build the Quad Torus

The presented way to build the quad torus is illustrative and explains all the Euler operators, but it is also somewhat tedious to follow. The full power of the Euler operator approach to building shape is revealed by slightly restructuring the construction process. Note that this results in the *same* object. It issues basically the same Euler operations, except a `makeVFS` instead of `makeEV` in step 6, and `killFmakeRH` except `killEmakeR` in step 9. The new process demonstrates how Euler operators can be grouped into functions, or higher-level operators, by reorganizing the operator sequence in a slightly different and more structured manner: There are many different sequences of Euler operators that result in the identical shape.

Turning a polygon into a double-sided face. A simple polygon without holes can be considered just a sequence p_0, p_1, \dots, p_n of points in 3D. It is converted into a double-sided polygon by a corresponding sequence of Euler operators that reads `makeVFS`, `makeEV(b)`, \dots , `makeEV(b)`, `makeEF`. There are $n - 1$ `makeEV` (b), one for each of the points p_1, \dots, p_n . The last operator closes the polygon and creates the second face. This is basically a generalization of steps 1–3 from the previous example. For creating the quad torus, this is applied twice, for the big and the small quad. Let f_B, b_B and f_S, b_S be the front and backsides of the big quad B and the small quad S , so that all sides are coplanar, and the front-sides f_B and f_S are facing into the same direction.

The extrude operation. Steps 4 and 5 can also be generalized to faces of arbitrary shape and degree, and also to faces with rings. This operation is called *extrude*. What happens is that for all vertices of a face (and of all rings), dangling vertices are created, in face normal direction. Subsequent dangling vertices (on all boundaries) are then connected via `makeEF` to create the side quads. Note that this basically amounts to a displacement of the original face, provided the side quads are all new faces. All the dangling vertices are created in the same distance d in the face normal direction. When d is negative, an ‘inverted’ shape can be created, with the CCW sides of all faces directed to the interior. For the quad torus, the extrude operation is applied to f_B with d , and to the backside b_S of S with $-d$.

Turning an inverted shape into a hole. The result is that the displaced face b'_S lies in the same plane as f'_B – but it has the opposite orientation, since the extrude operation does not affect the orientation of the face that is being displaced. The same holds for the front face f_S of S , which still lies in the same plane as b_B . So both faces f_S and b'_S of the small quad are ready to become rings of the respective faces b_B and f_B of the big quad, thus creating a hole through the object. The result is exactly the same object as the one shown in Fig. 2.15, step 12.

Practical relevance of Euler operators. The possibility to re-order an Euler sequence is also the deeper reason for the practical relevance of the Euler operators. It can be used for expressing the similarity of 3D objects, and it is also the actual foundation for the realization of the generative idea developed in the later chapters 4 and 3 of this thesis.

Operator	v	e	f	r	h	s
makeVFS	1		1			1
makeEV	1	1				
makeEF		1	1			
killEmakeR		-1		1		
killFmakeRH			-1	1	1	
<i>plane normal</i>	1	-1	1	-1	2	-2

$$M^{-1} = \frac{1}{12} \begin{pmatrix} 2 & 9 & -5 & 3 & -2 & 1 \\ -2 & 3 & 5 & -3 & 2 & -1 \\ 2 & -3 & 7 & 3 & -2 & 1 \\ -2 & 3 & 5 & 9 & 2 & -1 \\ 4 & -6 & 2 & -6 & 8 & 2 \\ 8 & -6 & -2 & -6 & 4 & -2 \end{pmatrix}$$

Figure 2.17: Algebraic properties of Euler Operators. Each operator creates or deletes a number of mesh entities, shown as rows in the table. The numbers can also be deduced from the operator names. The rows are linearly independent, spanning a 5-dimensional hyper-plane in signature space. The Euler-Poincaré equation (last row) is the 6th independent direction, which then amounts to a basis matrix. The inverse of this matrix (right) can be used to compute unique *Euler coordinates* from a given mesh signature.

2.2.3 Properties of Euler Operators

The Euler operators have an interesting relation to the extended Euler-Poincaré formula. With v, e, f being the number of vertices, edges, and faces of a mesh, and s, h, b, r the number of shells (or connected components), topological holes, border components, and rings, it reads

$$v - e + f = 2(s - h) - b + r ,$$

as it was stated in Theorem 2.27. With the Euler operators just presented, it is not possible to create meshes with border, so b is always 0. As was noted before, however, a border component always forms a closed loop, so it basically corresponds to a – not necessarily planar – face. A clean way to handle border components is thus to just flag such ‘border faces’ as faces that do not really belong to the mesh. These invisible faces are called *hollow faces*.

From the example in the last section it has become clear that the names of the individual operators correspond to the entities added to or removed from the mesh (see explanation of steps 4-5). The effect of the operators to the respective numbers from the Euler-Poincaré equation are listed in Fig. 2.17 (a). The vector of integer numbers (v, e, f, r, h, s) are called the *signature* of a mesh. The signature can be considered as a vector from a six-dimensional vector space. In this case each mesh corresponds to one vector, but of course different meshes can have the same signature. The application of an Euler operator transforms a mesh into another mesh and changes the signature. So a construction sequence, as in the quad torus example, can be considered as a path in 6-dimensional space. As there are only five directions to go, one for each Euler operator, the space that is spanned by the operators can at most have dimension five. And this is indeed the case, as the rows from Fig. 2.17 (a) are linearly independent when considered as 6-vectors. The inverse operators, killVFS etc., have a reversed signature, which is multiplied by -1, so they just lead to the same vector space.

One thing to note is that the second incarnation of the killFmakeRH operator, the killFsmakeR operator to join different shells, has a different signature, namely $(0, 0, -1, 1, 0, -1)$. It is not part of the above table, though, because it is not independent from the other operators: Instead of creating different shells that are subsequently joined into one connected component, it is as well possible to create the resulting object from only one shell. Another remark concerns the fact that table 2.17 (a) lists three constructive make... operators, but also two destructive ones: killEmakeR and killFmakeRH. This is due to what is perceived as the usual way

- to create rings, namely by detaching them from an outer boundary, as in step 9 of the quad torus example, and
- to creates holes, namely by turning faces into rings, as in step 12 of the quad torus example.

So the Euler operators span a 5-dimensional hyper-plane in 6-space. When the normal vector $n = (1, -1, 1, -1, 2, -2)$ of this hyperplane is added to the set of signature vectors, a full basis of the 6-dimensional space results. By adding it as another row to the table, an invertible 6×6 matrix M results; this matrix is just the right part of the table 2.17 (a). But a basis matrix is invertible, and the inverse matrix M^{-1} is shown in Fig. 2.17 (b). This matrix has the very interesting property that it allows to compute the ‘Euler coordinates’ of a given mesh. Take for example the quad torus from the previous section: It basically consists of two boxes, with the top and bottom faces of the inner box being rings of the respective faces from the outer box. The resulting numbers of mesh entities can be seen in the first table from Fig. 2.18, (a). The multiplication with M^{-1} yields

$$(16, 24, 10, 2, 1, 1) \cdot M^{-1} = (1, 15, 10, 1, 1, 0)$$

which is the vector that tells how many times each of the Euler operations have to be applied to obtain the mesh. This gives the second table in Fig. 2.18, (b). The fact that the last coordinate is 0 just means that the vector $(16, 24, 10, 2, 1, 1)$ is a valid combination of mesh entities, i.e., it fulfills the Euler-Poincaré equation.

v	16	makeVFS	1	1	makeVFS
e	24	makeEV	15	$v - 1$	makeEV
f	10	makeEF	10	$f + h - 1$	makeEF
r	2	killEmakeR	1	$r - h$	killEmakeR
h	1	killFmakeRH	1	h	killFmakeRH
s	1	<i>plane normal</i>	0		

Figure 2.18: Euler coordinates of a mesh. The Euler coordinates are the minimal number of Euler operators to create the mesh. Left boxes: The signature of the quad torus can be used to compute the respective (minimal) numbers of Euler operators needed to build it: There is a linear dependency between both integer vectors. The fact that the *plane normal* result is 0 means that the signature is valid (Euler-Poincaré). Note that the example construction uses exactly these numbers. Right: More general version for the signature $(v, e, f, r, h, 1)$, i.e., the minimal number of Euler operators for constructing any closed orientable manifold 2-complex that has only a single connected component ($s = 1$).

A more general result can be obtained for meshes with a single shell, i.e., when $s = 1$. The multiplication of the vector $(v, e, f, r, h, 1)$ with for instance the second column from M^{-1} yields the Euler coordinate of makeEV. The resulting expression can be drastically simplified using the Euler-Poincaré equation, by substituting $f = -v + e + r - 2h + 2$:

$$\frac{1}{12}(9v + 3e - 3f + 3r - 6h - 6) = \frac{1}{12}(9v + 3e - 3(-v + e + r - 2h + 2) + 3r - 6h - 6) = \frac{1}{12}(12v - 12) = v - 1$$

So this yields $v - 1$ as the Euler coordinate of makeEV, which is not such a surprise: besides makeVFS, makeEV is the only Euler operator that creates a vertex, so the total number of vertices should correlate with the number of times makeEV is to be applied. In a similar way, simple expressions for the other Euler coordinates can be obtained. They are summarized in the table to the right, in Fig. 2.18 (c). These numbers are minimal in that they represent the *least* number of operator applications. Note that they can also take negative values: High-genus objects, with $h > 0$, but without or with just a few rings, have $r - h < 0$. A negative number means that instead of killEmakeR, its inverse operator must be used. Thus, such objects cannot be constructed without use of the makeEkillR operator – which seems plausible.

2.2.4 Closedness and Completeness of Euler Operators

So far it has merely become clear that it is possible to construct *some* interesting surfaces using Euler operations. The fact that it is possible to compute Euler coordinates for every 2-complex that satisfies the Euler-Poincaré equation is also an indication for their general applicability. Yet there are many different meshes for each signature, and it is not so clear yet whether *any* possible 2-complex with rings can really be constructed using Euler operators. The second important issue concerns the question whether in some situations the application of an Euler operator can turn a 2-complex with rings into something that is not a 2-complex with rings. So the question is whether Euler operators are closed on the set of complexes on compact orientable surfaces of any genus. The following theorem clarifies on these questions. Its proof has two different parts, which will also be sketched, since they provide important insights in the nature of Euler operators.

Theorem 2.29 (Euler operators are closed and complete)

Let \mathcal{M} be the set of all closed orientable manifold 2-complexes with rings.

Then the Euler operators are a closed and complete set of operations for \mathcal{M} . More precisely:

- Any Euler operator that is applied to a mesh $m \in \mathcal{M}$ yields another mesh m' that is again in \mathcal{M} .
- Every mesh $m \in \mathcal{M}$ can be created by a finite sequence of Euler operators.

Proof part 1: Closedness of Euler operators. The first part of the theorem states that Euler operators are sound: It is not possible to make a mesh invalid by applying an Euler operator. An invalid mesh is a 2-complex that violates either one of the three criteria from the mesh consistency theorem 2.24, the manifold edge and vertex properties, and the orientability.

The theorem furthermore assumes that the operators are only used in a syntactically valid way: It is not legal, for instance, to use killEF to delete an edge that is incident to the same face on both sides; in this case, killEmakeR must be used, or killEV in case it is a dangling vertex. It is also illegal to delete a shell using killVFS if there are still some edges left. To check the conditions under which each of the operators can be applied is a subtle and important issue when implementing Euler operators.

The soundness must be proven for each operator and also its inverse operator, according to all the legal configurations that are shown in Fig. 2.13. The proof is inductive over the length of the sequence. So assume for the induction step that a closed orientable manifold 2-complex, i.e., a valid mesh, is given.

- **makeVFS/killVFS**

The first pair of operators, makeVFS and killVFS, trivially maintain the validity criteria, and so are sound.

- **makeEV/killEV**

The operators makeEV (a) and (b) create a dangling vertex, which just inserts an edge into the edge cycle of a vertex. The vertex split makeEV (c) does the same for two vertices. This does not affect the manifold vertex property, since no new cycles can be added to a vertex by makeEV; edges are only inserted in existing cycles. Neither can makeEV affect the manifold edge property, as all new edges are incident to exactly two faces. Orientability also still holds: It is not at all affected by inserting a dangling vertex with makeEV (a) and (b). Concerning makeEV (c), when all faces around a vertex are consistently oriented, this is also true if the vertex is split. So makeEV is sound.

Its inverse killEV only merges cyclic vertex lists, which preserves the manifold vertex property. The manifold edge property and the orientability are not affected either because only one edge vanishes. So killEV is also sound.

- **makeEF/killEF**

The operations makeEF (a), (b), and (c) are the topological duals of makeEV (a), (b), and (c). So one can argue that if makeEV is sound, then makeEF must also be sound since makeEV works like makeEF on the topological dual – and the dual of a closed orientable manifold 2-complex also has these properties. With the same argument, killEF must also be sound.

It is instructive to try checking the three criteria individually for makeEF. The duality between vertices and faces also helps to better understand the manifold vertex criterion. The cycle of edges around a vertex corresponds to a face boundary in the dual graph. The requirement of a single closed edge cycle for primal vertices carries over to the requirement of a single closed face boundary in the dual graph, i.e., a face in the dual graph may not have rings. A primal face with rings corresponds to a dual vertex with more than one edge cycle, which is a non-manifold, or *complex vertex* (see Fig. 2.1 and the table 2.28)

- **makeEkillR/killEmakeR**

The duality argument does not help with the soundness of the remaining two operators because they are using rings. The makeEkillR/killEmakeR pair of operators trades between rings and edges. But a 2-cell may not have multiple boundaries, since it cannot be homeomorphic to the 2-disc then. As mentioned before, this is repaired by taking every ring as if it was permanently connected with the outer face boundary. Irrespective of the number of rings, as well as their shape and position, it is always possible to introduce such ‘artifact edges’ in a way so that they do not cross (which is a consequence of the Jordan curve theorem, see [Kin93]), to restore the 2-disc property of a face with rings. Keeping this in mind, the makeEkillR/killEmakeR operators become irrelevant to the question of soundness.

- **makeFkillRH/killFmakeRH**

The last pair of operators, makeFkillRH and its inverse, are nothing but the connected sum of surfaces and the inverse operation, which disconnects a surface by cutting it along a closed cut path, filling in two small discs as plasters. The connected sum operation though is defined for surfaces, where it does not use rings. This is also possible using Euler operators: killFmakeRH turns a face into a ring of another face, and the ring can then be immediately removed using makeEkillR. So using the following operators in a sequence together, the connected sum is realized without rings.

- killFmakeRH + makeEkillR is the connected sum, to either join different shells into one, or to create a topological hole by joining a shell with itself.
- killEmakeR + makeFkillRH does the reverse by detaching a face that is connected over a bridge, i.e., a single edge, with the rest of the shell.

Concerning the consistency criteria the only critical issue is to maintain the orientability of the complex. The problem is to assert that a face and its ring always have opposite orientations. If this is not the case, a Klein bottle can seemingly be created, as shown in Fig. 2.19.

The connected sum affects the face signature in a simple way: Basically, one face of the mesh is traded for an additional edge, in order to create a topological hole. This is also compatible to the Euler-Poincaré equation, which reads $v - e + f = 2s - 2h$ in its simpler form: The connected sum increases the left-hand side of the equation by 2, since it becomes $v - (e + 1) + (f - 1)$, which has to be compensated by one fewer shell or one more hole on the right-hand side. Also note that the connected sum was actually defined only for surfaces, but it carries over to 2-complexes on the surfaces: In the surface case, two small discs are removed and the disc boundaries are identified. Correspondingly, on a 2-complex two faces can be removed, and their boundaries can be identified. This is possible, though, only if the boundaries are compatible, i.e., if they contain the same number of vertices and 1-cells. It is simpler, however, to set one face into the other, and to connect both with a bridge, which is the effect of using killFmakeRH + makeEkillR, as it was proposed.

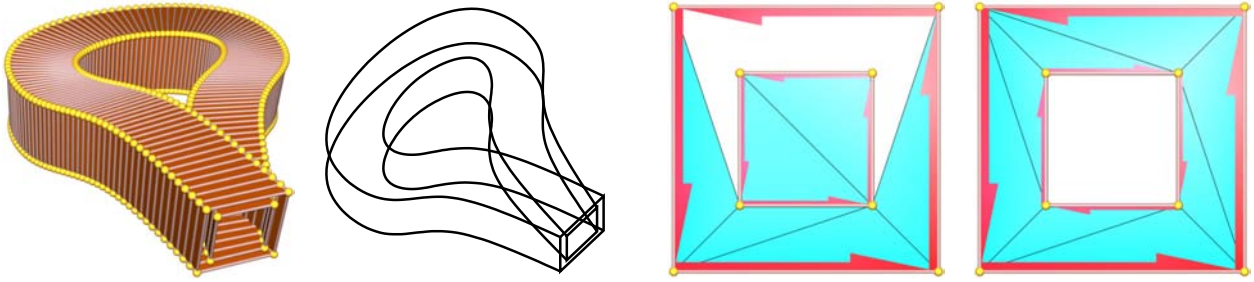


Figure 2.19: Problems with Klein Bottle and killFmakeRH. A long and thin rectangular box can be bent in two ways so that the ends meet: With the end faces facing each other, or facing in the same direction. While the first way creates a torus, the latter case leads to the Klein Bottle (a, b). But killFmakeRH is a topological operator, and it does not check the geometric validity of the configuration; consequently, it can be used regardless of how the box is bent and the faces are oriented. This can lead to the problem that the face interior is not well defined, in particular it cannot be triangulated correctly (c). For a valid triangulation (d), the face interior must always be to the *left* when traveling along the boundary, which implies a CCW orientation for the face boundary, but a CW orientation for rings.

Note that this is not a topological problem, but a problem with the embedding: a *geometrical pitfall* (see Fig. 2.28).

Proof part 2: Completeness of Euler operators. The completeness of Euler operators can be proven in a constructive way. This is interesting indeed, because it effectively gives an actual algorithm to convert a given mesh into a sequence of Euler operators. The underlying idea is fairly simple:

1. Given a valid mesh with rings, delete all of its entities,
2. record all Euler operators that were used for doing so, and finally
3. invert the sequence, by inverting each operator and reversing the sequence.

This algorithm relies on the invertibility of Euler operators. It terminates since a mesh has only a finite number of entities that can be removed. The question remaining is about the best order of operations. First, all rings can be removed in a simple way, by turning them into faces using makeFkillRH. This reflects the second view of rings, namely as temporally disabled faces: Faces that are not rendered for the moment, but that ‘lend’ their boundary to another face. So the face is still there, it is just simply marked as being a ‘ring’. With makeFkillRH, the mark is removed, and it becomes a face again.

Once there are no more rings, in principle any sequence of legal kill... operations will do: For instance, one could apply killEV and killEF as long as there are edges to do that legally. Edges with the same face on both sides can also be removed, creating a ring with killEmakeR, and the ring can be removed as already described. But according to the conditions from Fig. 2.14 there is a subtlety: If the edge has the same vertex on both ends, and also the same face on both sides, it is called a *double loop*; so it is not only incident to only a single vertex, but it is also adjacent to only a single face. Consequently a double loop corresponds also to a double loop in the dual graph. In this case, to apply killEmakeR is not legal – otherwise the face would become a ring of itself¹. Double loops may appear to be strange and very extraordinary – but the truth is that they have already appeared, for instance in the complexes on the torus in Fig. 2.5 and the double-torus in Fig. 2.11.

Instead of issuing operations in an unordered manner, the procedure EULERKILLMESH from Fig. 2.21 systematically first removes rings, then joins all faces (per shell), and retracts all vertices (also per shell). The result is a pointed sphere for shells with genus 0, a pair of double loops for genus 1 shells, two such pairs for genus 2, etc. So this procedure is effectively a constructive way to obtain the prototype surfaces from the surface classification theorem (Theorem 2.23): After step 3, there is only one face and only one vertex for a connected surface, and $2n$ edges, if n is the genus of the surface, just as promised by the table in Fig. 2.9. So one consequence of the theorem is that double loops use to come in pairs, which is why step 4 of the procedure removes two double loop edges at a time. The first of them is removed by the procedure KILLDOUBLELOOP (also in Fig. 2.21), which is a substitute for a specialized Euler operator. It basically works by temporarily adding one vertex, one edge, and one face, so that killEmakeR can be applied to the temporary edge. This is the green edge in Fig. 2.20 (c), which illustrates steps 1–5 of the procedure.

When all double loops are removed, all shells have been reduced to pointed spheres, so that they can simply be removed with killVFS, which is done in step 5 of EULERKILLMESH. This completes the sequence of Euler operators to remove the mesh, which can then be inverted to yield a sequence to construct the mesh.

¹Mäntylä [Män88] incorrectly applies killEmakeR to double loops in his `remedg` procedure for mesh removal (Program 16.10)

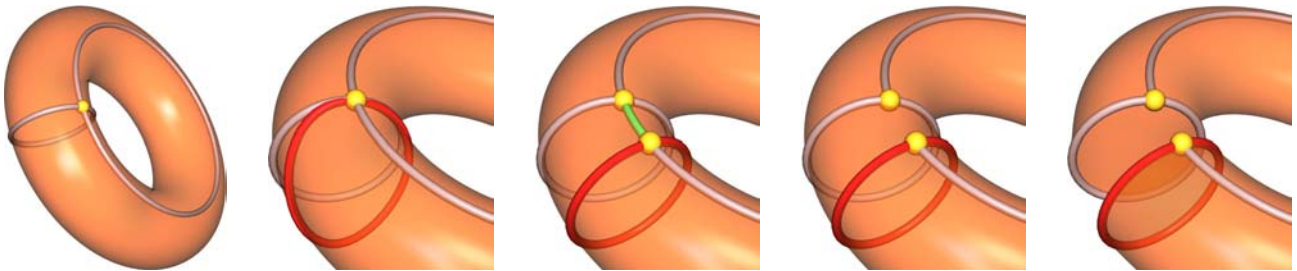


Figure 2.20: Removing a double loop. This shows steps 1–5 of procedure `KILLDOUBLELOOP` from Fig. 2.21.

`EULERKILLMESH`($m = (V, E, F)$)

- 1 **Remove all rings.** Turn all rings into faces using `makeFkillRH`. This increases the number of connected components, and it decreases the number of topological holes.
- 2 **Join all faces.** Apply `killEF` (c) as long as possible, i.e., to all edges with different faces on both sides. The result is that each shell has only a single face left, but possibly still many vertices.
- 3 **Collapse all edges.** For sphere-like shells (genus 0), the edge structure is now a tree. Apply `killEV` (b) and (c) to ‘retract’ this tree into one – arbitrarily chosen – root vertex. This leaves this vertex as the single vertex per shell.
- 4 **Remove all topological holes.** All remaining edges are *double loops*. Choose one and delete it with `KILLDOUBLELOOP`. Then the shell still has only one surface, but a new vertex v' was inserted to break a loop. Double loops come in pairs, so there is another edge that joins v and v' . Remove v' by collapsing this edge with `killEV`. Continue with the next double edge.
- 5 **Kill all shells** using `killVFS`.

`KILLDOUBLELOOP`(e)

- 1 Assert that edge e is a *double loop* that has the same vertex v on both ends and the same face f on both sides. There is no single Euler operator to remove such a double edge.
- 2 With `makeEF` (b), create a loop with edge l on v , creating a new face g_1 so that e and l are now both adjacent to f and g_1 .
- 3 Split v using `makeEV` (c) to separate the loops, so that l is now on the new vertex v' , e remains on v , and v and v' are connected with a new edge that is adjacent to g_1 on both sides.
- 4 Remove this edge with `killEmakeR` to make l boundary of a ring of the new face g_1 .
- 5 To remove the topological hole, turn this ring into a face g_2 of its own using `makeFkillR`.
- 6 To clean up, note that the boundaries of the new faces g_1 and g_2 are loops, attached to v and v' , respectively. So g_1 and g_2 can be removed with $2 \times \text{killEF}$.

Figure 2.21: Mesh Removal using Euler operators. Left: Steps 1–3 of procedure `EULERKILLMESH` produce the prototype surfaces from Fig. 2.9. What remains are *double loops* forming the skeleton of topological holes. They can be removed a pair at a time (step 4), using the procedure `KILLDOUBLELOOP` to the right (see Fig. 2.20).

2.3 What is a Mesh? – A Definition

Later chapters in this thesis are very much based on meshes, so it is vital to make unambiguously clear what a mesh is. In section 2.1, the term *mesh* was introduced, and preliminarily defined as being synonymous to valid two-dimensional cell complexes. The following definition is the promised attempt to describe more precisely what is generally used and known as *meshes* in computer graphics.

2.3.1 Mesh Definition

The following definition of a mesh is supposed to be compatible to 2-complexes on the one hand (as from the definition of cell complexes 2.15 for $n = 2$), and to practical meshes on the other. Two-complexes and manifold surfaces are formally well-defined concepts. But neither of them exactly corresponds to the computer graphics notion of meshes. It is not easy to define meshes, though, because their use is very heterogeneous. But only some form of definition permits to theorize about the exact nature of the differences mentioned.

Definition 2.30 (Mesh)

A mesh consists of three finite sets V, E, F , which are the vertices, edges, and faces of an abstract 2-complex, and an embedding σ . The embedding maps all mesh entities, i.e., all vertices, edges, and faces, to subsets of \mathbb{R}^3 in a way that

- the embedding $\sigma(v)$ of each vertex $v \in V$ is a 0-cell,
- the embedding $\sigma(e)$ of each edge $e \in E$ is a 1-cell, and
- the embedding $\sigma(f)$ of each face $f \in F$ is a 2-cell.

Vertices, edges, and faces are tied together by their boundaries. This is understood as an abstract incidence relationship, and is called the connectivity of the mesh:

- The boundary of an edge $e \in E$ consists of two end vertices $v, v' \in V$. v and v' may be the same.
- The boundary of a face $f \in F$ is an alternating cyclic sequence $(v_0, e_0, v_1, e_1, \dots, v_{n-1}, e_{n-1})$, $n \geq 1$, of vertices and edges, where v_i, v_{i+1} have to be the end vertices of e_i .

Each vertex may appear only once in a face boundary. Since they are cyclic sequences, indices are modulo n , and edge e_{n-1} goes from v_{n-1} to v_0 . Faces may have multiple boundaries, in which case one of them is denoted as base face, and the others as rings. Every edge must be part of at least one face boundary, every vertex must be part of at least one edge.

The embedding must be compatible to the incidence relationship: The boundary of $\sigma(e) \subset \mathbb{R}^3$ must be the points $\sigma(v), \sigma(v') \in \mathbb{R}^3$. For a face boundary, the sequence $(\sigma(v_0), \sigma(e_0), \sigma(v_1), \sigma(e_1), \dots, \sigma(v_{n-1}), \sigma(e_{n-1}))$ of the respective embeddings must be continuous and identical to the boundary of $\sigma(f)$. The boundary sequence determines the face orientation.

This is a weaker version of the definition of 2-complexes, particularly with respect to the identity of cells and their embedding: A 2-complex in 3-space is made of cells that are 0-, 1- and 2-dimensional open subsets of \mathbb{R}^3 . Some of the cells touch at their borders, and this defines the incidence relationship. Meshes in computer graphics are defined the other way around: Vertices, edges, and faces are *abstract* entities that are not identical with 0-, 1-, and 2-cells, as it is the case with a 2-complex. Instead, they are only *mapped* to cells. But each cell can be mapped to by multiple abstract entities. This has important consequences, for example

- different vertices can be at the same position in space,
- different edges may occupy the same 1-cell (as subset of \mathbb{R}^3),
- faces that touch along their boundaries do not have to be glued together, and
- the cell interiors do not have to be disjoint, so edges may cross faces, and different faces may be stuck in each other.

When a pair of 2-cells of a 2-complex mutually intersect, a 1-cell is automatically required along the intersection curve. And when a 1-cell hits through a 2-cell there has to be an explicit 0-cell at the hit point. The mesh definition is less restrictive since it requires only the individual embeddings to be cells, i.e., homeomorphic to the 0-, 1-, or 2-disc, and thus free of self-intersections. To require all faces to have pairwise disjoint interiors, as 2-complexes do, would unnecessarily rule out many meshes used in practice. From a computer graphics point of view, self-intersections are not too much of a problem for many algorithms. The z -buffer algorithm, for instance, is robust against mutually intersecting faces, so that a mesh can in all cases be rendered correctly by the graphics hardware. The only notable exception are co-planar faces with different colors which exhibit the well-known z -buffer artifacts. – The separation between the ‘abstract’ complex and its embedding reflects two different aspects of a mesh, one of them discrete, the other of continuous nature:

- The discrete *connectivity* is defined by the incidence relationship on the entities of the abstract complex (V, E, F) .
- The ‘continuous’ *geometry* of a mesh is defined by its embedding σ .

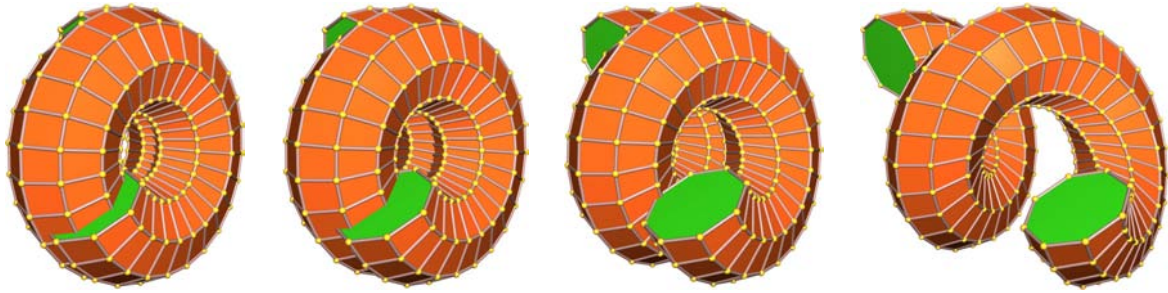


Figure 2.22: Self-intersections are sometimes hard to detect. This illustrates the difference between theory (cell complexes) and practice (meshes): All objects have the same connectivity, but only the rightmost object (d) is a cell complex; the first three objects contain self-intersections, and the mapping from \mathbb{R}^3 to the cells is not unique. All objects shown are part of the same continuous family of parametric objects. It would be unnatural, and computationally expensive, to restrict the stretch parameter only to the interval corresponding to valid cell complexes. This is an important argument justifying the use of meshes instead of cell complexes in computer graphics. The objects are primarily composed of quadrangles, but note that none of them is planar. This is an example of another issue mentioned: where exactly is the surface of higher-degree faces located? The practical answer is that faces with degree > 3 are triangulated. So the ‘true’ surface can be regarded as an artifact of the triangulation algorithm. What the graphics hardware eventually displays is only a collection of simplicial 2-cells: triangles.

This separation is motivated by the way computers work. The fundamental underlying problem is the very particular ‘psychology’ of a computer, which is quite different from a mathematician’s human perception, and also from mathematical formalism: It requires special effort in terms of developing algorithms, as well as in computing resources (space and time) to actually make a computer realize that there is such thing as a self-intersection in a given 3D object. Being part of computer science, computer graphics inherits a particular approach to solve practical problems, namely to prefer – possibly even many – local solutions over one global solution, where ‘everything’ relates to ‘everything’.

One of the consequences of the separation is, for instance, that the embedding can be changed without changing the connectivity. Vice versa this is not possible, due to the required continuity of the embedding of the face boundaries: When for example an edge is added to split a face, σ has to account for the new edge and face, and it must be modified for the face that is split. Another consequence is that the criteria for mesh consistency, as formulated for 2-complexes in theorem 2.24, must be modified in order to tell whether a given mesh realizes a closed orientable manifold surface. According to the separation between abstract complex and embedding, it is split into two parts. They tell whether a mesh *is* a closed orientable manifold, or whether it *could be* one, only with a different embedding.

Definition 2.31 (Topological Consistency)

A mesh is topologically consistent if and only if an embedding σ exists so that the mesh is a closed orientable manifold surface. Equivalently, the abstract complex must satisfy the three criteria from the mesh consistency theorem 2.24,

- the manifold edge property,
- the manifold vertex property,
- and it must be orientable.

To also allow meshes with borders, the manifold edge property can be weakened, so that every edge must only be incident to ≤ 2 edges, and border vertices may have a single ‘open’ edge cycle.

It is important to note that the three consistency criteria in theorem 2.24 are purely topologic in nature, since the embedding is irrelevant for them: The consistency can be verified by checking only the ‘discrete’ incidence relationships between the entities – which, in the case of 2-complexes, are in perfect synchronization with the ‘continuous’ embedding.

Definition 2.32 (Geometrical Consistency)

A mesh is geometrically consistent if and only if it is a closed orientable manifold surface. That is, it is topologically consistent, and the embedding σ is reversible, i.e., for all cells, σ^{-1} is a single unique abstract entity.

While topological consistency can be maintained very efficiently, it is usually much more difficult to assure geometric consistency. In order to see what can go wrong, it is best to look at some concrete cases. Quite a typical example to illustrate the difference between cell complexes and meshes is shown in Fig. 2.22. It shows the same abstract complex with four different embeddings only one of which is a cell complex. – One very strong incentive for the mesh definition 2.30 came from the necessity to make it compatible with meshes represented as indexed face sets.

2.3.2 General Meshes in Computer Graphics: Indexed Face Sets

Many data structures, algorithms, and mesh file formats in computer graphics use *indexed face sets* (IFS's) for the description of 3D surfaces. They are very general in that they are just based on an index scheme, for instance, an enumeration of the different entities. This imposes the fewest restrictions in terms of geometric or topological consistency. There are many different specialized versions of indexed formats, but they usually have a few things in common.

Definition 2.33 (Indexed Face Set)

An indexed face set is a mesh with the following additional properties:

- The embedding of each edge is a straight line segment.
- An edge $e \in E$ is uniquely identifiable by an (unordered) pair of end vertices v, v' , which formally establishes an equivalence relation: $(v, v') \sim (v', v) \sim e$.

Its concrete representation is a set of lists, each for a different type of data. These are usually

- vertex positions, as a list (p_1, p_2, \dots, p_n) of n 3D points $p_i = (x_i, y_i, z_i)$
- normal vectors, as a list (n_1, n_2, \dots, n_k) of k 3D vectors $n_i = (x_i, y_i, z_i)$
- texture coordinates, as a list (t_1, t_2, \dots, t_l) of l 2D points $t_i = (u_i, v_i)$, and
- faces, as a list (b_0, b_1, \dots, b_m) of m face boundaries. Each face boundary b_i is again a (cyclic) list, with variable length m_i , of index tuples, one for each vertex on the face boundary.
- Vertex tuples may contain indices of vertex positions, normal vectors, and texture coordinates, in a specified order, some of which can be optional.

Where exactly is the surface of an indexed face? This is a particularly problematic issue concerning the face definition. It only prescribes the face boundary to be a piecewise straight, closed polygon in \mathbb{R}^3 , basically a sampled curve in space. But by no means it has to be planar. So it is only clear that the surface is suspended 'somewhere' in the face border, but there is no obvious, canonical, definition of the surface. More formally, there is not always a well-defined continuous mapping from the unit 2-disc to \mathbb{R}^3 so that the disc boundary matches the face boundary polygon. Highly non-planar boundaries may be exceptional, but one has to keep in mind that computers realize a finite-precision floating point universe: The fourth point of a quad may simply not be representable so as to make the quad *exactly* planar – just because no suitable point exists in the discretized space. And non-planarity again raises the question of the exact surface definition. The tolerancing issues that are inevitable with higher-degree faces in arbitrary position can be avoided by using exclusively triangles. A subclass of indexed face sets are the indexed triangle sets. They are basically IFS's that contain only degree 3 faces, and they are sometimes also called *triangle soups*: Just a collection of triangles in space.

The importance of file formats. The indexed face set approach is used by a number of mesh file formats, such as VRML, the .obj format from Wavefront Inc., the .off ('object file format'), and others. Despite the fact that all these formats use the same underlying approach, there are subtle differences between them. This can be the reason for an inevitable loss of information when converting from one format to another: The target format may not be able to represent some of the data of the source format, or the target format may require additional data that cannot be derived from the source file. The similarity between two of the ascii formats for indexed face sets, .obj and VRML, is demonstrated in Fig. 2.23. The VRML format is based on different types of *nodes*, each with different *fields*, such as the coord field for the coordinate list of the IndexedFaceSet node in the example. Using fields, additional meta-information can be specified, such as the material, whether faces have to be triangulated, or whether back-faces can be culled away because the object is solid. Also the orientation can be specified using the boolean ccw field to determine which side is outside.

Both formats VRML and .obj permit faces of arbitrary degree, but they do not support faces with rings. It is possible though to specify normal vectors and texture coordinates per vertex. In .obj, this is done with lines of the form $n\ x\ y\ z$ and $t\ u\ v$. Normals and texture coordinates have an ID just like vertices, which can be used when specifying a face: $f\ v_0/t_0/n_0\ v_1/t_1/n_1\ \dots\ v_k/t_k/n_k$ is the more general form of faces in .obj, where the ID triplet $v_i/t_i/n_i$ specifies which vertex position, texture coordinate, and normal is to be used for vertex i of this face.

2.3.3 Are Meshes necessary for Rendering? – The OpenGL Answer

A whole spectrum of indexed geometry descriptions exists. Indexed face sets, such as .obj and VRML, are quite redundant compared to today's efficient mesh encoding schemes [TG98]. Redundancy is not tolerable when efficiency and performance are primary goals, for instance, when communicating with the graphics hardware over the display driver. Realtime requirements dictate that a new image has to appear 20-30 times per second. So the question is: What are

```

# Wavefront .obj file example
# =====
v -1.25 -1.25 -1.25 # V1: 000
v -1.25 -1.25 1.25 # V2: 001
v -1.25 1.25 -1.25 # V3: 010
v -1.25 1.25 1.25 # V4: 011
v 1.25 -1.25 -1.25 # V5: 100
v 1.25 -1.25 1.25 # V6: 101
v 1.25 1.25 -1.25 # V7: 110
v 1.25 1.25 1.25 # V8: 111
v 2.00 0.50 0.50 # V9
f 1 2 4 3 # F1: 0**
# replace 5 7 8 6 F2 : 1**
f 5 7 9 # F2a: 1**
f 7 8 9 # F2b: 1**
f 8 6 9 # F2c: 1**
f 6 5 9 # F2d: 1**
f 4 8 7 3 # F3: *0*
f 2 1 5 6 # F4: *1*
f 2 6 8 4 # F5: **0
f 1 3 7 5 # F6: **1

# VRML indexed face set example
# =====
Shape {
  appearance Appearance {
    material Material { diffuseColor 1 1 0 }
  }
  geometry IndexedFaceSet {
    coord Coordinate {
      point [
        -1.25 -1.25 -1.25, # V0: 000
        -1.25 -1.25 1.25, # V1: 001
        -1.25 1.25 -1.25, # V2: 010
        -1.25 1.25 1.25, # V3: 011
        1.25 -1.25 -1.25, # V4: 100
        1.25 -1.25 1.25, # V5: 101
        1.25 1.25 -1.25, # V6: 110
        1.25 1.25 1.25, # V7: 111
        2.00 0.50 0.50 # V8
      ]
    }
    coordIndex [
      0,1,3,2,-1, # F1 : 0**
      # replace 4,6,7,5 # F2 : 1**
      4, 6, 8,-1, # F2a: 1**
      6, 7, 8,-1, # F2b: 1**
      7, 5, 8,-1, # F2c: 1**
      5, 4, 8,-1, # F2d: 1**
      3,7,6,2,-1, # F3 : *0*
      1,0,4,5,-1, # F4 : *1*
      1,5,7,3,-1, # F5 : **0
      0,2,6,4,-1, # F6 : **1
    ]
  }
  creaseAngle 0.0
  convex TRUE
  solid TRUE
  colorPerVertex TRUE
  ccw TRUE
}

```

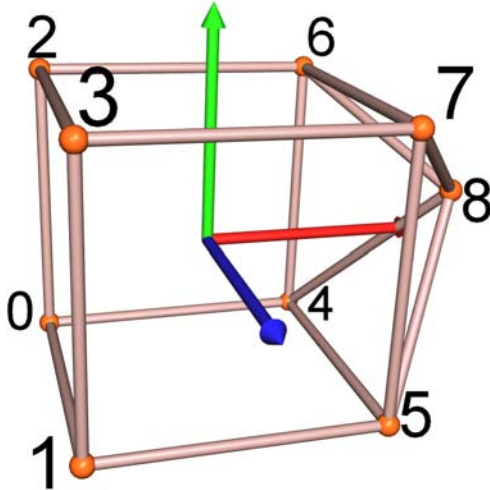


Figure 2.23: Indexed Face Set file formats. Two times the same object is shown, in the .obj format from Wavefront Inc. (left) and in VRML syntax (right). Faces are given as index lists with indices referring to the vertex sequence. Vertices are implicitly numbered by their order, starting at 0 (VRML) or at 1 (.obj).

```

GLfloat vertex [] = { -1.25,-1.25,-1.25, -1.25,-1.25,+1.25,
  -1.25,+1.25,-1.25, -1.25,+1.25,+1.25,
  +1.25,-1.25,-1.25, +1.25,-1.25,+1.25,
  +1.25,+1.25,-1.25, +1.25,+1.25,+1.25, +2.00,+0.50,+0.50 };
GLuint tristrip [] = { 4,0,6,2,7,3,5,1,4,0 };
GLuint quad [] = { 0,1,3,2 };
GLuint trifan [] = { 8,7,5,4,6,7 };
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT,0,vertex);
glDrawElements(GL_TRIANGLE_STRIP, 10, GL_UNSIGNED_INT, tristrip);
glDrawElements(GL_TRIANGLE_FAN, 6, GL_UNSIGNED_INT, trifan);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_INT, quad);
glDisableClientState(GL_VERTEX_ARRAY);

```

Figure 2.24: Indexed Face Set in OpenGL. The object from Fig. 2.23 can be rendered with three display primitives from Fig. 2.25: A quad for the bottom (0,1,3,2), a triangle fan around vertex 8, and a triangle strip for the rest.

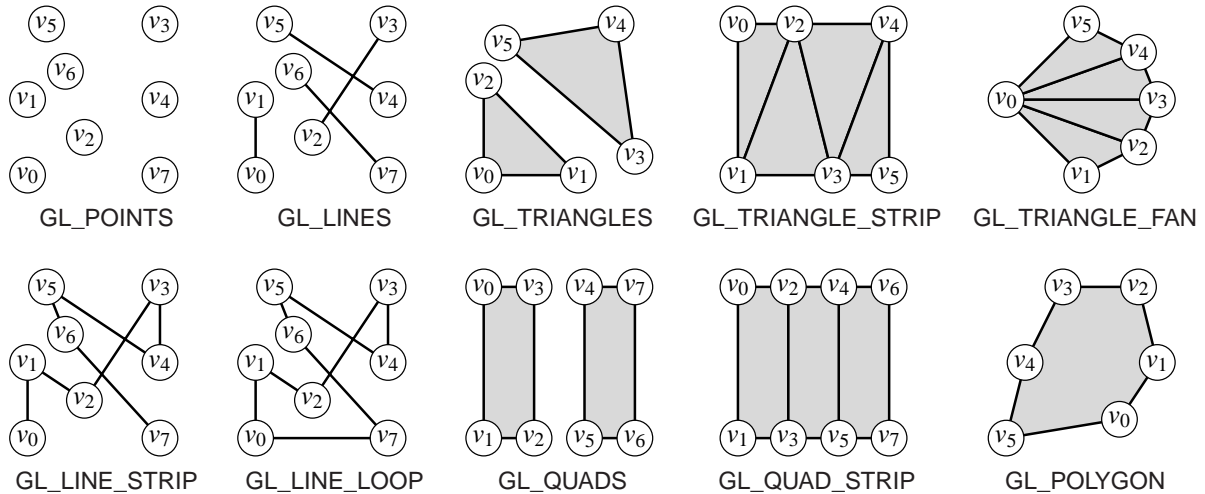


Figure 2.25: OpenGL display primitives. A vertex sequence is partitioned according to the chosen primitive type. For `GL_TRIANGLES` and `GL_QUADS`, vertex triplets and quadruples correspond to triangles and quads. In a triangle strip, every single new vertex v_i forms a new triangle v_{i-1}, v_{i-2}, v_i with the previous two vertices from the sequence, which are swapped every other time for a consistent orientation. This happens also with quad strips, where a pair v_i, v_{i+1} of new vertices is swapped to form the quad $v_{i-2}, v_{i-1}, v_{i+1}, v_i$ with the previous pair. With a triangle fan, the first vertex v_0 is special, since every new vertex v_i forms a triangle v_0, v_{i-1}, v_i . The polygon primitive is a bit redundant, since the specification says that a polygon must be convex, so it can be rendered with a triangle fan.

suitable low-level graphics display primitives? The trade-off here is between changeability and compactness of the mesh description.

The answer to this question given by the OpenGL graphics standard [WND97]: Meshes are not needed for rendering. Instead of meshes, OpenGL uses the ten graphics primitives shown in Fig. 2.25. For surfaces it supports isolated triangles, quadrangles, and convex polygons. They directly correspond to the vertex and face lists from the IFS definition 2.33, as for each degree n face a sequence of n vertices is to be transmitted, $n = 3$ or 4 . The vertex sequence may be a continuous stream of (x, y, z) values for v_0, v_1, v_2, \dots . This is inefficient when the same vertex is used several times. The solution is to use *vertex arrays*: The vertex sequence is specified by its beginning in main memory, i.e., a pointer to the start of an array. Individual array elements can then be referenced simply by their index, so that multiple references create not so much overhead. Efficiency can be gained by grouping primitives together by their types.

To achieve more space efficiency, OpenGL offers also *strip geometry*: With triangle strips and fans, every newly arriving vertex creates one new triangle, by re-using the two previous vertices. Two indices have to be swapped for a

lower border insertion	0	1	2	3	4	5	4	6	7	8	9	
			012		234		454		467		789	
				213		435		456		768		
upper border insertion	0	1	2	3	4	3	5	6	7	8	9	
			012		234		433		567		789	
				213		433		536		768		
triangle fan emulation	0	1	0	2	0	3	0	4	0	5		
			010		020		030		040			
				012		023		034		045		

Figure 2.26: Swapping OpenGL triangle strips. In the middle of each row, the upper row shows the strip indices, the next two rows show the triangles created. For every other triangle, two indices are swapped (lower row). OpenGL does not render empty triangles, which is the case when two of three indices are equal (shown in bold). This can be used for inserting additional vertices on the upper or lower border of a strip, for vertices with higher valence than 4. This principle can be extended to the point where a triangle fan can be rendered as a strip.

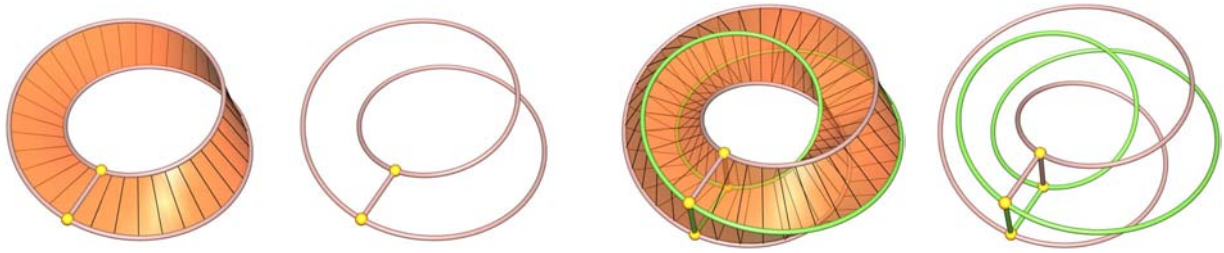


Figure 2.27: Inflating the Möbius band. When the Möbius band has a thickness, it becomes a torus-like object with a rectangular profile. But before gluing the ends together, one end is rotated by 180 degrees. Remarkably, the result is an orientable object! It has two sides but only two boundary curves, both of which can be seen as boundary curves of a Möbius band.

consistent orientation of every other triangle (Fig. 2.25). Note that there is not much difference between triangle and quad strips, since the first can be transformed into the latter by joining pairs of triangles, and both use an identical indexing scheme. Triangle fans can also be reduced to triangle strips, as shown in Fig. 2.26, so the triangle strip is an efficient and versatile geometric primitive. There are even practitioners who consider a list of triangle strips (TSL, for *triangle strip list*) as the *only* geometry primitive really necessary, disregarding fully fledged polygonal meshes to be a waste of time and space. A 3D object stored in vertex arrays can be rendered with just a few lines of C code, as demonstrated in the code example in Fig. 2.24. It is comprised essentially of specifying two pieces of information:

- **vertex data:** memory locations for the vertex positions with `glVertexPointer` and a normal array, color array, etc.
- **index arrays:** a primitive type (enum) and the location of an array with indices, using `glDrawElements`.

Remarkably OpenGL too realizes the separation between connectivity and embedding from the mesh definition 2.30.

Consistency issues, pitfalls, and mesh repair. What is appealing about IFS's is that they are very general, because they can represent any collection of surface pieces: Triangles, quadrangles, and isolated polygons in space. So any type of surface can be represented, even if it has borders, is non-orientable, not connected, and so on. The generality is of course also a drawback, since a priori not much is known about a given mesh in IFS format. The term 'face set' stresses the fact that global consistency in any way is not enforced. A few of the flaws that can occur when a mesh is not well defined are listed in Fig. 2.28. Some of these flaws can be more easily remedied than others, which opens the field for *mesh repair algorithms*.

Remarkably, there are just three topological pitfalls, which correspond to the three mesh consistency criteria, but quite a bit can go wrong with the geometry of a mesh. What is even worse, most of the geometric problems can only be checked approximately, and some sort of 'epsilon' is involved because of the finite floating point resolution. Connectivity problems on the other hand can be detected clearly and efficiently.

2.3.4 Mesh Manipulation with Euler Operators

A mesh that is represented as indexed face set can not be changed very easily. Unlike geometry changes, which can be done by moving the vertices, changes of the connectivity are not so simple. Indexed face sets do not store neighbourhood information explicitly which would be needed for connectivity changes.

As an example, consider the cuboidal object from Figs. 2.23 and 2.24. Suppose one wishes to insert the midpoint of edge (1,5) as a new vertex with index 9 in the VRML description. The edge is part of face (1,5,7,3) that then has to become (1,9,5,7,3). To insert it only in this face, however, would make it a T-vertex. So it must also be inserted in the face (1,0,4,5) that contains the back-edge (5,1). But to find this face requires to search through all edges of all faces. So this simple vertex insertion is an $O(n)$ operation for an indexed face set, just because indexed face sets do not store explicit neighbourhood information.

The elegance of Euler operators. But connectivity changes are not only a matter of computational complexity, and not only a question on the level of data structures and mesh implementations. The question always arises: Which set of operations shall be implemented for modeling? – The Euler operators offer a conceptually clean way to manipulate manifold meshes on the lowest level, i. e., on the level of individual vertices, edges, and faces. On the one hand, they are a theoretical device to prove and reason about properties of 2-complexes. The great advantage of Euler operators is that they consistently handle all special cases: Isolated vertices, dangling edges, rings, and faces that are neighbouring on themselves.

Topological Pitfalls

<i>complex vertex</i>	A non-manifold vertex of the abstract complex. So with any embedding there is more than one single edge cycle for it, or the edge cycle is not closed. The different surfaces in its neighbourhood are just not homeomorphic to the 2-disc.
<i>complex edge, edge with multiple sheet</i>	A non-manifold edge that has more than two faces attached to it. Complex edges of solids have an even number of sheet, and the topological consistency can be restored by splitting a complex edge up into multiple manifold edges.
<i>non-orientable edge</i>	An edge that has two faces attached to it, but with opposite orientations, so that both face boundaries traverse the edge in the same direction – unlike orientable edges, that are traversed in both directions by the two boundaries.

Geometrical Pitfalls

<i>self-intersection</i>	General term for any violation of the condition 2 for cell complexes (Def. 2.15), which requires that $\text{Int}(\sigma) \cap \text{Int}(\tau) = \emptyset$ for every pair σ and τ of cells.
<i>touching vertex</i>	A vertex that lies in the interior of an edge or a face of a different part of the surface, so that the surface touches itself.
<i>T-vertex</i>	A vertex v incident to two edges e_1, e_2 , that touches another edge e_3 . Furthermore, both edges e_1, e_2 also lie on e , but they are not identified with it. A T-vertex lies on the boundary of e_3 's face, but is not part of it, so it is actually a border vertex of the surface.
<i>multiple vertex</i>	Several different vertices at the same position in space, for instance, a pair $v \neq w$ with $\sigma(v) = \sigma(w)$. They are sometimes added intentionally, e.g., to remove a complex vertex.
<i>null edge, zero-length edge</i>	A pair of unidentified vertices with an edge between them, so the face degree is seemingly wrong. Zero length edges are used intentionally, e.g., when triangular Bézier patches are required but only quadrangular patches are available.
<i>loop, degree 1 face</i>	While routinely used in topology where edges can bend, the geometry of a loop is a face collapsed to a single point when using only straight edges. So special provision is to be taken for defining the loop geometry.
<i>degree 2 face</i>	While routinely used in topology where edges can bend, a degree 2 face collapses to a line when using only straight edges. Degree 2 faces are used together with null edges, for instance, when splitting a solid with a plane (see for instance the solid splitting algorithm from Mäntylä, [Män88] in chapter 14).
<i>multiple edges</i>	Several different edges with the same embedding, so both end vertices are multiple vertices.
<i>non-planar face</i>	A face where no plane in \mathbb{R}^3 contains all vertices of the face boundary, not even when allowing a ‘tolerable’ error.
<i>reversed orientation</i>	A shell or just a single face that is ‘inside out’, because the face orientation is not compatible. Easy to check and to repair for closed meshes (interior must be finite), not so easy for meshes with border.
<i>non-simple face boundary</i>	The boundary polygon lies in a plane, but it intersects itself. This can happen when exchanging successive vertices which, for instance, turns a quad into a ‘bowtie’.
<i>outside ring</i>	A ring that is not (completely) contained within the boundary of its base face. Sometimes caused by confused ring and base face.
<i>non-coplanar ring</i>	Typically used for a planar face that is made a ring of another planar face, but unfortunately the planes are (very) different.
<i>reversed ring</i>	A problem caused by a ring that is actually <i>not</i> reversed, i.e., when the ring has the same orientation as the base face. May be caused by an attempt to create a non-orientable object, such as the Klein bottle (Fig. 2.19).

Figure 2.28: Possible Pitfalls with Polygonal Meshes.

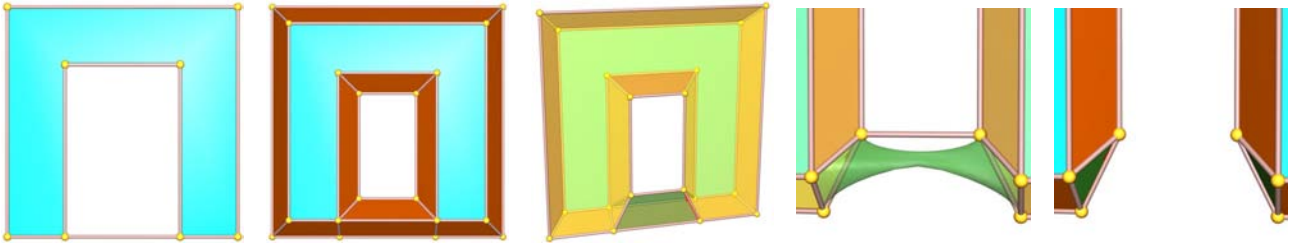


Figure 2.29: Inflating neighbouring face and ring. Two vertices are set into the lower edge of a double-sided quad, provided with dangling vertices using $2 \times \text{makeEV}$ (b), and connected with makeEF . The new face is then made a ring of the backside quad using killFmakeRH . Now the mid lower edge ‘looks’ non-manifold, since it is adjacent to the backface on one side and to a ring of the backside on the other (a). This situation can be clarified by “inflating” the backside using an expanding extrude operation (b). The ring can be removed by inserting two edges on the back using $\text{makeEkillR} + \text{makeEF}$, creating the green face (c) to remove the ring. Removing two edges with $2 \times \text{killIEF}$ basically restores the situation in (a), leaving one edge to connect two loops, so that the green face has degree 8 (d). The remaining edge can be removed by $\text{killEmakeRH} + \text{makeFkillR}$, leaving the loops as two triangle faces (d). Note that without inflating, these faces have degree 1, i.e., they are loops.

A particularly useful feature of Euler operators is that they can handle faces of any degree. But sometimes this generality may not be needed, e.g., when using only triangle meshes. Then the Euler sequence for constructing these meshes can have a simpler structure, because it turns out that sequences with always the same few operations are used. In this case it makes sense to combine these operations, and to set up a new, derived, set of mesh construction operations with them, replacing the Euler operators. A notable example for this approach are the edge collapse / vertex split operations that will be introduced in section 4.1.6 (Fig. 4.10): A vertex split is nothing but $\text{makeEV}, \text{makeEF}, \text{makeEF}$.

2.3.5 A Note on Seemingly inconsistent Mesh Configurations

With Euler operators every manifold mesh can be realized, and at the same time the connectivity of the mesh is guaranteed to remain valid at all times (theorem 2.29). But one has to bear in mind that this concerns only the validity of the abstract complex and not necessarily a valid embedding. Edges of a mesh are usually only straight line segments. Consequently, a loop will always be invisible, since it collapses to a straight line of zero length, i.e., to a point. But note that the reason of such problems is only an inadequate embedding.

When practically using Euler operators for building meshes, it is not always easily possible to figure out what the problem is with a given embedding. Most of the problems, e.g., the pitfalls from Fig. 2.28 cannot be solved only by visual inspection of a model. One solution to this problem is to *inflate* inflate a model the model. Formally, to inflate a model means to subdivide and expand it around the critical parts, in order to make the embedding more distinct with the added geometry. When the solution to the problem is found, it can in most cases be transferred back to the unexpanded solution, due to the algebraic properties of Euler operators. One example where inflation is quite useful is to make a loop edge visible by expanding it to, e.g., a circle. This can be done by a sequence of makeEV . This technique was used, for instance, in Fig. 2.20 that shows how to remove a double loop. The solution that was found for this problem could be translated back to the unexpanded circles, i.e., to the case of loop edges.

Using inflation, even orientability problems can be clarified. The prototype of a non-orientable surface is the Möbius band which contains a surface with only one side. Surprisingly, an orientable surface can be created by inflating it, as shown in Fig. 2.27. The images of the Möbius band shown earlier in this chapter are actually ‘fakes’: They were all just created by collapsing one of the two surfaces of the inflated Möbius band.

Another seemingly non-manifold configuration can be created by a face that is turned into a ring of a direct neighbour face. This introduces an edge that is a non-manifold edge – seemingly. But this edge does *not* affect the integrity of the mesh: As demonstrated in Fig. 2.29, inflation reveals that the edge can be removed by a sequence of $\text{makeEkillR} + \text{makeEF} + \text{killEmakeR} + \text{makeFkillRH}$ operations. The first two operators remove the ring and create loops, i.e., degree 1-faces, which are then disconnected. To clean up, the loops can then be removed with $2 \times \text{killIEF}$. It may seem strange to connect a vertex with itself using makeEkillR , but this is nevertheless a legal operation.

This example illustrates the fact that Euler operators are better thought of as abstract mesh manipulation operators, than as user-friendly modeling tools. User-friendly modeling tools should rather be based on Euler operators.

2.4 Hierarchical Meshes for Interactive Modeling and Visualization

The goal of this thesis is a system for interactive inspection and on-line manipulation of three-dimensional models represented by surface meshes. Note that the latter requires the first: Long display latencies are unacceptable when constructing a 3D object with the aid of a computer. A good modeler must incorporate an efficient viewer. The considerations so far have revealed two different aspects of meshes, relating to both of these issues interactive visualisation and modeling.

- **Euler operators permit the consistent manipulation of manifold meshes.**

Euler operators reflect a paradigm change from objects to operations: A mesh is understood as the result of a sequence of parameterized operations rather than a set of unrelated faces. A particular sequence of Euler operations can be re-used: It will always reliably reproduce the same type of connectivity changes, provided it is used under the same specified conditions. A fixed operator sequence can still be parameterized, namely in terms of the vertex positions. Even more power have variable operator sequences: The extrude operation (section 2.2.2) is a notable example of a variable sequence with a clear structure: $n \times \text{makeEV}$, then $n \times \text{makeEF}$, to create a degree n face.

The downside is that the neighbourhood of a face and a vertex must be readily accessible, ideally in constant time, for Euler operators to be efficient. Another drawback is the limitation to manifold meshes.

- **Indexed face sets are an efficient and general low-level mesh representation.**

Indexed face sets, in particular indexed triangle sets organized as triangle strip lists, are a very space efficient surface representation. Triangle strips are compatible to low-level graphics APIs such as OpenGL, and today's graphics acceleration boards are optimized for displaying large triangle meshes at interactive rates. Triangles are 2-simplices, and they permit to approximate any kind of 2D surfaces embedded in 3D at any desired resolution.

Indexed face sets do not store explicit face neighbourhood information. The reason is that for the purpose of mesh display, neighbourhood information is not needed, so at this point it is just an unwanted overhead. It is indispensable though for consistent mesh manipulation, since changing the connectivity implies to change face neighbourhoods.

Neither of both views on meshes is entirely satisfactory. But it is possible to combine them in a way such that they mutually compensate for their deficiencies and their advantages are added. The solution is to use them in a hierarchical fashion with the *patch complex* approach.

Definition 2.34 (Patch, Patch Complex, Tesselation)

A generally non-planar 2-cell $P \subset \mathbb{R}^3$ is called a *patch*. If the mapping from a parameter space $U \subset \mathbb{R}^2$ to P is explicitly given, then P is called a *parametric patch*.

A patch can be approximated by simplicial subdivision, the result is called a *tesselation*. A tesselation is a 2-complex with border, of course homeomorphic to the 2-disc, made of triangles whose vertices are sampled from the patch.

A patch complex is a 2-complex with non-planar faces. So its faces are patches, and its edges, where adjacent patches meet, are in general curves in 3-space.

The distinguishing feature of a patch complex, one that makes it different from, e.g., a polyhedron, is its embedding: The faces are essentially polygons in space, but they can be arbitrarily bent and stretched. The term 'patch' is also used in a more sloppy fashion, in that also planar faces are sometimes called patches, especially when they have curved borders. The patch complex examples in Fig. 2.30 demonstrate the idea of *hierarchical meshes*: The surface is partitioned into curved or flat patches, and each of the patches is then tesselated.

- **High-level modeling with Euler operators.**

Euler operators manipulate only the abstract complex of a mesh. The embedding of each face is a whole patch. This reduces the overhead of storing connectivity information with faces, since there are *much* fewer faces in the mesh than in the tesselation. A great shape complexity can be achieved with relatively few faces that have a very detailed tesselation. So mesh faces become higher-level data structures, some sort of mechanism for grouping many low-level simplices together. This opens also new possibilities, e.g., for culling²: To cull away individual triangles is inefficient, because it is faster to just render them, whereas culling of whole patches is worthwhile.

- **Low-level display with indexed face sets.**

The tesselation is directly stored as indexed face set (triangle strip list), so that it can be processed by the graphics hardware in an optimal way. The patch representation can, and should, be chosen in a way that the computations are very schematic, so that they can be highly optimized. For each patch, also different levels of detail of the tesselation can be computed. This makes it possible to adapt the surface resolution both to the position of the viewer, and to the capabilities of the machine (Fig. 2.30 (c) and (g)), which is called *adaptive rendering*.

²Culling means to detect objects that are not visible, so that they do not have to be processed by the rendering pipeline.

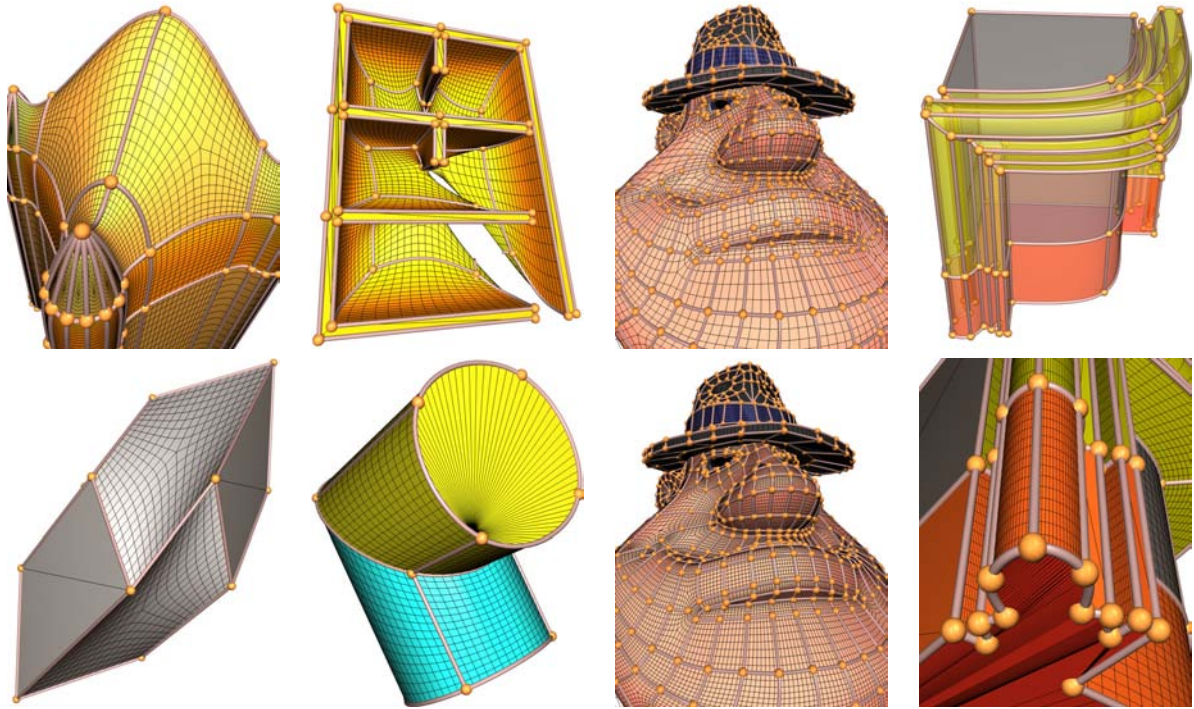


Figure 2.30: Patch complex examples.

The big question remaining is how to define the patches. In fact a great number of possible surface representation schemes exists, from Hermite patches over Bézier- and B-spline surfaces, to, most notably, NURBS. When choosing a particular patch type for a patch complex, one important requirement is that adjacent patches can be seamlessly and smoothly stitched together. Methods exist to achieve smooth transitions, for instance, between trimmed NURBS patches, but they are usually computationally very expensive due to the underlying theory of *geometric continuity*. Good overviews over the different possibilities to define free-form patches and the methods to stitch them together is given in the books from Farin [Far02] and Hoschek and Lasser [HL92].

On the other hand, not all shapes are exclusively made of smooth, curved surface parts. Many objects, such as for example buildings, are composed of planar polygons for the most part – but they exhibit important free-form features in prominent places on their façades. So a method was required that integrates well with polygonal meshes on the one hand, but also permits to use curved patches within the same data structure. In particular, smooth surfaces should not require much overhead in terms of degrees-of-freedom with respect to the polygonal parts of the surface.

As a result of these considerations, *subdivision surfaces* have been chosen as the representation for smooth surfaces.

Chapter 3

Subdivision Surfaces

This chapter introduces Catmull/Clark subdivision surfaces. It shows their origin from B-Spline surfaces, introduces limit rules, discusses their behaviour in practice, and illustrates some typical problems; finally it develops an optimized method for fast interactive display. The treatment is somewhat verbose because Catmull/Clark surfaces are of great importance for this thesis. They have a number of unique features:

- **Subdivision surfaces make free-form modeling manageable**

Only very few degrees of freedom must be specified to create appealing smooth free-form shapes. Meaningful changes can be applied in an intuitive way but the surface remains always smooth and connected. The method is extremely robust and computationally stable.

- **Subdivision surfaces are compatible with polygonal meshes**

Every manifold polygonal mesh can be used as control mesh for subdivision surfaces. In particular the same data structure can (and will) be used for both ways of modeling. And even more importantly, the same type of operations, Euler operators, can be used for both polygonal and free-form modeling.

- **Subdivision surfaces support optimized interactive display**

The complexity of the control mesh is typically two or more orders of complexity smaller than the tessellation. Subdivision surfaces are inherently multi-resolution surfaces, with a strictly hierarchical structure so no further computations are needed to change the resolution. The evaluation can be greatly optimized so that more than 50K 9×9 -patches can be computed per second on an average PC.

A short historic overview The basic idea of recursive subdivision is to refine a given initial mesh, the *control mesh*, by inserting new vertices, edges, and faces on every level, according to a number of rules, the *subdivision rules*. They are designed in a way that in the limit, when the number of subdivision steps goes to infinity, the mesh converges to a smooth surface that possesses a continuous tangent plane everywhere.

The idea of recursive subdivision has been first applied to curves: In 1974 Chaikin [Cha74] presented his idea of successively refining a polygon that converges to a curve. Subdivision surfaces have then been introduced four years later, as early as 1978, in the two landmark papers *Recursively generated B-Spline Surfaces on Arbitrary Topological Meshes* from Ed Catmull and James Clark, and *Behaviour of Recursive Division Surfaces near Extraordinary Points* from D. Doo and Malcolm Sabin. Both appeared in the same famous issue 6, vol. 10, of *Computer-Aided Design* [CC78, DS78].

The *Catmull/Clark scheme* proceeds by splitting all faces of a given mesh recursively into smaller quadrangular sub-faces; it was designed as a generalization of bicubic B-spline surfaces. They are therefore C^2 continuous almost everywhere, except at irregular vertices, where they are at least tangent plane continuous. The *Doo/Sabin scheme* replaces with each refinement step (i) a degree m -face by a smaller degree m -face, (ii) a valence n -vertex by a degree n -face, and (iii) each edge by a quad. The Doo-Sabin surfaces are a generalization of biquadratic B-splines and, thus, C^1 -continuous almost everywhere. Another nine years later, in 1987, Charles Loop eventually discovered a subdivision scheme for triangle meshes. The *Loop scheme* recursively splits a triangle in four sub-triangles by inserting a new vertex on every edge of the mesh. This scheme generalizes quartic triangular B-splines.

It is probably fair to say that subdivision surfaces were not a very hot research topic for the longest time. In any case they have been put back into focus by the article *Subdivision Surfaces in Character Animation* from Tony DeRose, Michael Kass and Tien Truong that appeared on Siggraph 1998 [TDT98] together with the amazing short film *Geri's game* produced by Pixar. The subdivision surface approach proved so useful that within a few years, dozens of different subdivision schemes and modifications have appeared. Sources for an overview over existing schemes are the Siggraph course notes [ZS99] and the books from Joe Warren and Cohen et al. [WW02, CRE01].

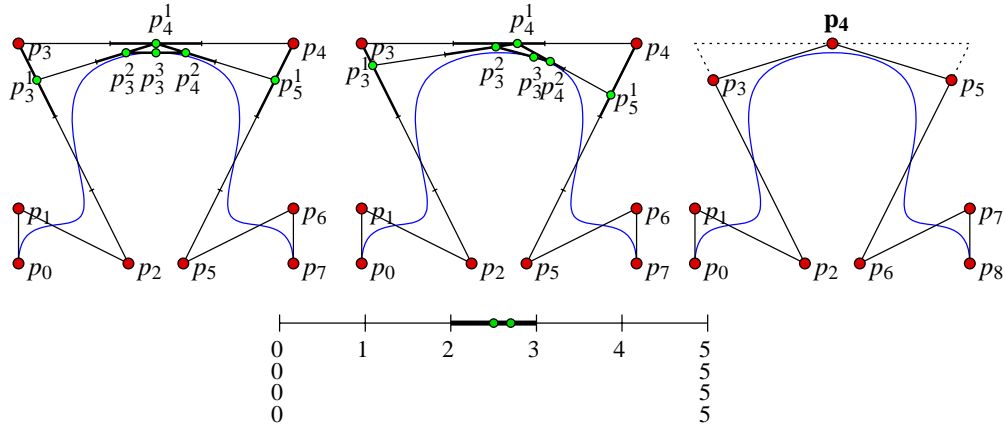


Figure 3.1: Knot intervals mapped to B-spline control polygon.

3.1 Genesis of the Catmull/Clark Scheme

The basic idea from Catmull and Clark was to generalize the popular bicubic uniform B-spline surfaces, so that every manifold mesh could serve as control mesh, irrespective of the vertex degree. B-spline surfaces are tensor product surfaces. This is a general technique to create surface representations from curve representations in basis form. This gives good control over the shape, but the tensor product approach requires control points to be arranged by rows and columns. The control mesh must therefore be a regular grid where all interior control vertices, also called CVs, have valence four. So the story of Catmull/Clark surfaces actually begins with the famous B-spline basis functions.

Definition 3.1 (B-spline functions) Let (t_0, t_1, \dots, t_k) be a nondecreasing sequence of scalar values, called a knot vector. The B-spline functions N_i^n of degree n , with $1 \leq n \leq k - 1$ and $0 \leq i \leq k - n - 1$, are recursively defined as follows. As knot values are allowed to coincide, the result of a division by zero is defined as zero, in particular $\frac{0}{0} := 0$.

$$N_i^0(\mathbf{t}) = \mathbf{1}_{[t_i, t_{i+1}]}(\mathbf{t})$$

$$N_i^n(\mathbf{t}) = \frac{\mathbf{t} - t_i}{t_{i+n} - t_i} N_i^{n-1}(\mathbf{t}) + \frac{t_{i+n+1} - \mathbf{t}}{t_{i+n+1} - t_{i+1}} N_{i+1}^{n-1}(\mathbf{t})$$

The basis of the recursion are the box functions $\mathbf{1}_{[a,b]}(\mathbf{t})$ of an interval $[a, b]$, which are equal to 1 if $a \leq \mathbf{t} \leq b$ and equal to 0 everywhere else. The recursive definition of the basis functions can be understood in different ways, for instance as a convolution, as in the Siggraph 99 course notes [ZS99]. The reason for the fractional form of the coefficients is that they are the relative position of \mathbf{t} in the interval $[a, b]$ with respect to start and end of the interval according to the mappings

$$\mathbf{t} \mapsto \frac{\mathbf{t} - a}{b - a}, \quad \mathbf{t} \mapsto 1 - \frac{\mathbf{t} - a}{b - a} = \frac{b - \mathbf{t}}{b - a} .$$

To facilitate this interpretation, \mathbf{t} is printed in bold in the above definition; and Fig. 3.1 shows a graphical interpretation on the segments of the control polygon. The B-spline functions have many interesting properties, the most important are:

- Compact support, inherited from the box functions: N_i^n is equal to zero outside the interval $[t_i, t_{i+n+1}]$.
- The B-spline functions are piecewise polynomials with a monomial representation for each interval.
- Functions N_i^n are C^{n-1} , but only if successive knot values are different.
- In case j knot values coincide, the respective B-spline functions are only C^{n-j} .
- The functions form a partition of unity $\sum_{i=0}^{k-n-1} N_i^n(\mathbf{t}) = 1$ on the inner intervals, i.e., for all $\mathbf{t} \in [t_n, t_{k-n}]$.

The knot vector defines the interval sizes. In case of equally spaced knot values the knot vector is called *equidistant*. In the simplest case, an equidistant knot vector is just a sequence of integers like for instance $(-3, -2, -1, 0, 1, 2, 3, 4)$. By careful inspection of the diagram in Fig. 3.3 the properties mentioned above can be verified.

As the B-spline functions form a partition of unity they can be used as weight functions. All linear combinations with respect to these weights yield an affine combination of the coefficients, for any $\mathbf{t} \in [t_n, t_{k-n}]$. But note that the coefficients can not only be numbers, they can be taken from any affine or vector space. In such a case the coefficients are called *control vertices* or short CVs. Given a knot vector (t_0, \dots, t_k) there are $k - n$ B-spline functions of degree n that can be used with a control polygon of the same size to form a curve, an example for which is shown in Fig. 3.1

$$\begin{array}{ll}
 N_0^0 = \mathbf{1}_{[-3,-2]} & N_1^0 = \mathbf{1}_{[-2,-1]} \\
 N_2^0 = \mathbf{1}_{[-1,0]} & N_3^0 = \mathbf{1}_{[0,1]} \\
 N_4^0 = \mathbf{1}_{[1,2]} & N_5^0 = \mathbf{1}_{[2,3]} \\
 N_6^0 = \mathbf{1}_{[3,4]} & \\
 \end{array}
 \qquad
 \begin{array}{l}
 N_0^1(t) = (-t-1)N_1^0 + (t+3)N_0^0 \\
 N_1^1(t) = -tN_2^0 + (t+2)N_1^0 \\
 N_2^1(t) = (t+1)N_2^0 + (-t+1)N_3^0 \\
 N_3^1(t) = tN_3^0 + (-t+2)N_4^0 \\
 N_4^1(t) = (t-1)N_4^0 + (-t+3)N_5^0 \\
 N_5^1(t) = (t-2)N_5^0 + (-t+4)N_6^0
 \end{array}$$

Figure 3.2: B-spline functions of degrees 0 and 1 for the equidistant knot vector $(-3, -2, -1, 0, 1, 2, 3, 4)$. The highlighted box function N_3^0 isolates the middle interval $[0, 1]$. The graphs of these functions are shown in Fig. 3.3.

$$\begin{array}{l}
 N_0^2(t) = \frac{1}{2}((t^2+6t+9)N_0^0 + (-2t^2-6t-3)N_1^0 + t^2N_2^0) \\
 N_1^2(t) = \frac{1}{2}((t^2+4t+4)N_1^0 + (-2t^2-2t+1)N_2^0 + (t^2-2t+1)N_3^0) \\
 N_2^2(t) = \frac{1}{2}((t^2+2t+1)N_2^0 + (-2t^2+2t+1)N_3^0 + (t^2-4t+4)N_4^0) \\
 N_3^2(t) = \frac{1}{2}(t^2N_3^0 + (-2t^2+6t-3)N_4^0 + (t^2-6t+9)N_5^0) \\
 N_4^2(t) = \frac{1}{2}((t^2-2t+1)N_4^0 + (-2t^2+10t-11)N_5^0 + (t^2-8t+16)N_6^0) \\
 \\
 N_0^3(t) = \frac{1}{6}((t^3+9t^2+27t+27)N_0^0 + (-3t^3-15t^2-21t-5)N_1^0 \\
 \quad + (3t^3+3t^2-3t+1)N_2^0 + (-t^3+3t^2-3t+1)N_3^0) \\
 N_1^3(t) = \frac{1}{6}((t^3+6t^2+12t+8)N_1^0 + (-3t^3-6t^2+4)N_2^0 \\
 \quad + (3t^3-6t^2+4)N_3^0 + (-t^3+6t^2-12t+8)N_4^0) \\
 N_2^3(t) = \frac{1}{6}((t^3+3t^2+3t+1)N_2^0 + (-3t^3+3t^2+3t+1)N_3^0 \\
 \quad + (3t^3-15t^2+21t-5)N_4^0 + (-t^3+9t^2-27t+27)N_5^0) \\
 N_3^3(t) = \frac{1}{6}(t^3N_3^0 + (-3t^3+12t^2-12t+4)N_4^0 \\
 \quad + (3t^3-24t^2+60t-44)N_5^0 + (-t^3+12t^2-48t+64)N_6^0)
 \end{array}$$

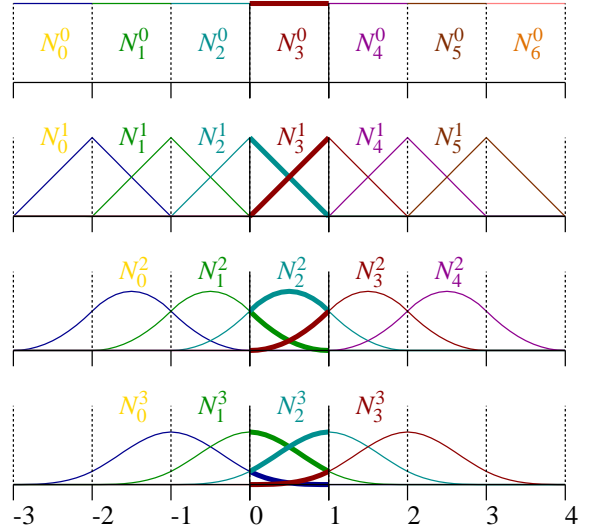


Figure 3.3: B-spline functions.

Definition 3.2 (B-spline curve) Let (t_0, t_1, \dots, t_k) be a knot vector, n a number such that $1 \leq n \leq k-1$, and P a control polygon with $k-n$ control vertices, so $P = (p_0, \dots, p_{k-n-1})$. The B-spline curve c of degree n for $\mathbf{t} \in [t_n, t_{k-n}]$ is then defined as

$$c(\mathbf{t}) = \sum_{i=0}^{k-n-1} p_i N_i^n(\mathbf{t})$$

Some properties of this curve can again be derived from inspection of the diagram in Fig. 3.3 for the equidistant example. For $\mathbf{t} = 0$ for example, it is $N_1^2(0) = N_2^2(0) = 0.5$, while the other basis functions are zero. Accordingly the curve for $n = 2$ is exactly in the middle between p_1 and p_2 at this time.

3.1.1 Evaluating points on a B-spline curve

Especially for curves with many CVs it is quite inefficient to evaluate all basis functions separately. A more efficient method make use of the compact support: For $\mathbf{t} \in [t_i, t_{i+1}]$ only the basis functions N_{i-n}, \dots, N_i are nonzero. Furthermore the recursion can proceed by combining the control points directly, rather than computing weights. This constitutes the de Boor algorithm for B-spline curve evaluation shown in Fig. 3.4.

To make the processing very clear the de Boor-algorithm for the important cubic case is explicitly listed in Fig. 3.5. It is identical to the general algorithm, but it has the loops for $n = 3$ unrolled. Only a sub-polygon of the CVs of length four is needed to compute the curve in any given sub-interval $[t_i, t_{i+1}]$, namely the four points $(p_{i-3}, p_{i-2}, p_{i-1}, p_i)$. Just as with Bézier curves, new points are computed that lie on the line segments of the control polygon. And just as with Bézier curves they make up a new polygon so that every interpolation step yields one fewer point:

$$(p_{i-3}, p_{i-2}, p_{i-1}, p_i) = (p_{i-3}^0, p_{i-2}^0, p_{i-1}^0, p_i^0) \longrightarrow (p_{i-2}^1, p_{i-1}^1, p_i^1) \longrightarrow (p_{i-1}^2, p_i^2) \longrightarrow p_i^3 = c(t)$$

The difference to Bézier curves, though, is that the curve parameter t is not related to the interval $[0, 1]$ but to an interval that is defined by the first and last knots from a (sliding) knot sub-sequence of the knot vector.

```

DEBOOR(t, k, n, (t0, ..., tk), (p0, ..., pk-n-1))
1 Find i such that t ∈ [ti, ti+1]
2 (p00, ..., pk-n-10) ← (p0, ..., pk-n-1)
3 for j ← 1 to n
4   do for m ← i - n + j to i
5     do s ← (t - tm) / (tm+n+1-j - tm)
6       pmj ← (1 - s) pm-1j-1 + s pmj-1
7 return pin

```

Figure 3.4: De Boor algorithm, general version.

```

DEBOOR-CUBIC(t, k, (t0, ..., tk), (p0, ..., pk-n-1))
1 Find i such that t ∈ [ti, ti+1]
2 (p00, ..., pk-40) ← (p0, ..., pk-4)
3 s ← (t - ti-2) / (ti+1 - ti-2)   pi-21 ← (1 - s) pi-30 + s pi-20
4 s ← (t - ti-1) / (ti+2 - ti-1)   pi-11 ← (1 - s) pi-20 + s pi-10
5 s ← (t - ti-0) / (ti+3 - ti-0)   pi-01 ← (1 - s) pi-10 + s pi-00
6 s ← (t - ti-1) / (ti+1 - ti-1)   pi-12 ← (1 - s) pi-21 + s pi-11
7 s ← (t - ti-0) / (ti+2 - ti-0)   pi-02 ← (1 - s) pi-11 + s pi-01
8 s ← (t - ti-0) / (ti+1 - ti-0)   pi-03 ← (1 - s) pi-12 + s pi-02
9 return pi3

```

Figure 3.5: De Boor algorithm, cubic version.

This is visualized in Fig. 3.1, at the beginning of the section, using another equidistant knot vector. This time the *multiplicity* of the first and last knot values is four, which makes the spline curve interpolate the end points. The diagram demonstrates the evaluation of the cubic B-spline curve at parameter values 2.5 or 2.7. These values reside in the knot interval $[t_5, t_6] = [2, 3]$, and the respective points on the curve are computed from the sub-polygon (p_2, p_3, p_4, p_5) . The position of the parameter is set in relation to an interval “window” of four, three, and two knot values at the different levels of the de Boor-algorithm.

By evaluating the spline curves at all relevant parameter values the curve can be sampled and thus converted to an approximating polygon. Yet such an approximation can also be derived from the control polygon directly. One possibility is *knot insertion*, as shown in Fig. 3.1 (c). By repeatedly inserting new knots to the knot sequence the control polygon is also refined while still representing the same curve. At the same time the control polygon follows the curve more tightly: In principle, infinitely repeated knot insertion yields a control polygon that is identical to the curve.

3.1.2 Recursive Subdivision of B-spline Curves

Catmull and Clark had another idea, though, for refining the control polygon, namely through *subdivision*. The main idea is to have a look at the uniform case of a cubic B-spline curve, especially with the knot vector from Fig. 3.3, $(t_0, \dots, t_7) = (-3, -2, -1, 0, 1, 2, 3, 4)$. B-spline curve are piecewise polynomials that are masked out using the box functions N_i^0 . The box function of the unit interval $[0, 1] = [t_3, t_4]$ is N_3^0 , which is printed in bold in the Figure. For $\mathbf{t} \in [0, 1]$, the B-spline curve can thus be written in matrix form like this:

$$\begin{aligned}
c(\mathbf{t}) &= \sum_{i=0}^3 p_i N_i^n(\mathbf{t}) = (p_0, p_1, p_2, p_3) \begin{pmatrix} N_0^3(\mathbf{t}) \\ N_1^3(\mathbf{t}) \\ N_2^3(\mathbf{t}) \\ N_3^3(\mathbf{t}) \end{pmatrix} = (p_0, p_1, p_2, p_3) \cdot \frac{1}{6} \begin{pmatrix} (-t^3 + 3t^2 - 3t + 1) \mathbf{N}_3^0(\mathbf{t}) \\ (3t^3 - 6t^2 + 4) \mathbf{N}_3^0(\mathbf{t}) \\ (-3t^3 + 3t^2 + 3t + 1) \mathbf{N}_3^0(\mathbf{t}) \\ (t^3) \mathbf{N}_3^0(\mathbf{t}) \end{pmatrix} \\
&= (p_0, p_1, p_2, p_3) \cdot \frac{1}{6} \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 4 & \\ -3 & 3 & 3 & 1 \\ 1 & & & \end{pmatrix} \begin{pmatrix} \mathbf{t}^3 \\ \mathbf{t}^2 \\ \mathbf{t} \\ 1 \end{pmatrix} =: P^T M^T T
\end{aligned}$$

In this notation, $P = (p_0, p_1, p_2, p_3)^T$ is the (column) vector of control vertices, $T = (\mathbf{t}^3, \mathbf{t}^2, \mathbf{t}, 1)^T$ are the monomial basis functions, and M is the basis change matrix for changing from the monomial basis to the B-spline basis. Note that M is actually the transpose of the 4×4 matrix above. Now given this curve over $[0, 1]$, consider to split it in halves so that one sub-curve c' runs over $[0, \frac{1}{2}]$, and another sub-curve c'' runs over $[\frac{1}{2}, 1]$. But the first sub-curve c' shall actually have a parametrization over $[0, 1]$, so that its parameter \mathbf{t}' travels twice as fast as the parameter of the original curve c . This can be expressed through a matrix S to obtain c' directly from c :

$$c'(\mathbf{t}') = (p_0, p_1, p_2, p_3) \cdot \frac{1}{6} \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 4 & \\ -3 & 3 & 3 & 1 \\ 1 & & & \end{pmatrix} \begin{pmatrix} \frac{1}{8} \\ \frac{1}{4} \\ \frac{1}{2} \\ 1 \end{pmatrix} \begin{pmatrix} \mathbf{t}'^3 \\ \mathbf{t}'^2 \\ \mathbf{t}' \\ 1 \end{pmatrix} =: P^T M^T S^T T$$

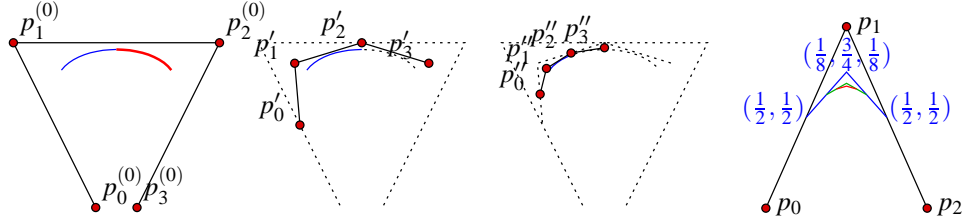


Figure 3.6: B-spline subdivision. (a) The original curve over $[0, 1]$ with its four CVs is split in two parts over the sub-intervals $[0, 0.5]$ and $[0.5, 1]$. (b) The CVs p'_i for the left part of the curve. (c) Repeated subdivision yields the CVs for the $[0, 0.25]$ part of the original curve. (d) Situation around a vertex.

On the other hand, c' is just a cubic polynomial curve – so it must be possible to express it as a B-spline segment. In particular, there must exist four control vertices $P' = (p'_0, \dots, p'_3)$ for c' so that it can be written as follows:

$$c'(t') = P'^T M^T T$$

How do the points P' relate to the original CVs P ? The transposed versions of the above equations are $c'(t) = T^T S M P$ and $c'(t) = T^T M P'$. They can only be true for all infinitely many values of t' if the following holds:

$$M P' = S M P \iff P' = M^{-1} S M P = (M^{-1} S M) \cdot P =: H \cdot P$$

The inverse of M exists, and the matrix multiplication to obtain $H = M^{-1} S M$ yields:

$$H = \frac{1}{18} \begin{pmatrix} 2 & -3 & 3 \\ -1 & 3 \\ 2 & 3 & 3 \\ 18 & 11 & 6 & 3 \end{pmatrix} \cdot \frac{1}{8} \begin{pmatrix} 1 \\ 2 \\ 4 \\ 8 \end{pmatrix} \cdot \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 \\ -3 & 3 \\ 1 & 4 & 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & \frac{1}{8} \\ \frac{1}{2} & \frac{3}{4} & \frac{1}{8} \\ \frac{1}{2} & \frac{1}{2} & \frac{3}{4} & \frac{1}{8} \\ \frac{1}{2} & \frac{1}{2} & \frac{3}{4} & \frac{1}{8} \end{pmatrix}$$

Here M is not transposed, unlike above. – The result of this computation, H , is quite specular, actually. The new CVs relate to the old ones in a suprisingly simple way, they are linear combinations:

$$P' = \begin{pmatrix} p'_0 \\ p'_1 \\ p'_2 \\ p'_3 \end{pmatrix} = \frac{1}{8} \begin{pmatrix} 4 & 4 \\ 1 & 6 & 1 \\ 4 & 4 \\ 1 & 6 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{pmatrix} = \begin{pmatrix} \frac{1}{2}(p_0 + p_1) \\ \frac{1}{8}(p_0 + 6p_1 + p_2) \\ \frac{1}{2}(p_1 + p_2) \\ \frac{1}{8}(p_1 + 6p_2 + p_3) \end{pmatrix}$$

The result of repeated subdivision of the control polygon is depicted in Fig. 3.6. The sequence (a)-(c) shows how the control polygon gets closer and closer to the curve with repeated refinement. But also note that the number of CVs essentially doubles every step. Yet the process is quite efficient: (d) shows only three refinement steps (in blue, green, red), and a fourth step would already be hard to see. Also note that subdivision has a *smoothing* component: The original vertices in 3.6 (d) form an acute angle that is widened in every step – this is not very surprising, though, as in the limit, the process needs to converge to a smooth curve.

An interesting thing to note is that there are only two different kinds of new vertices: Those that lie in the middle between two old vertices, with weights $(\frac{1}{2}, \frac{1}{2})$, are called *edge points*, and those that are a weighted sum of three consecutive vertices, with the weights $(\frac{1}{8}, \frac{6}{8}, \frac{1}{8})$, are *vertex points*. As shown in Fig. 3.6 (d), three vertices are sufficient to perform subdivision around a single vertex. According to the construction, the vertex must eventually converge to a point on the B-spline curve – but is there a faster way to compute the limit position than performing an infinite number of refinements steps? Let K be the upper-left 3×3 sub-matrix of H , then the situation at a vertex point is as follows:

$$\begin{aligned} P' &= \begin{pmatrix} p'_0 \\ p'_1 \\ p'_2 \end{pmatrix} = \frac{1}{8} \begin{pmatrix} 4 & 4 \\ 1 & 6 & 1 \\ 4 & 4 \end{pmatrix} \cdot \begin{pmatrix} p_0 \\ p_1 \\ p_2 \end{pmatrix} = K \cdot P \\ P'' &= K \cdot P' = K \cdot K \cdot P \\ P^{(n)} &= (K^n) \cdot P \\ P^\infty &= (\lim_{n \rightarrow \infty} K^n) \cdot P \end{aligned}$$

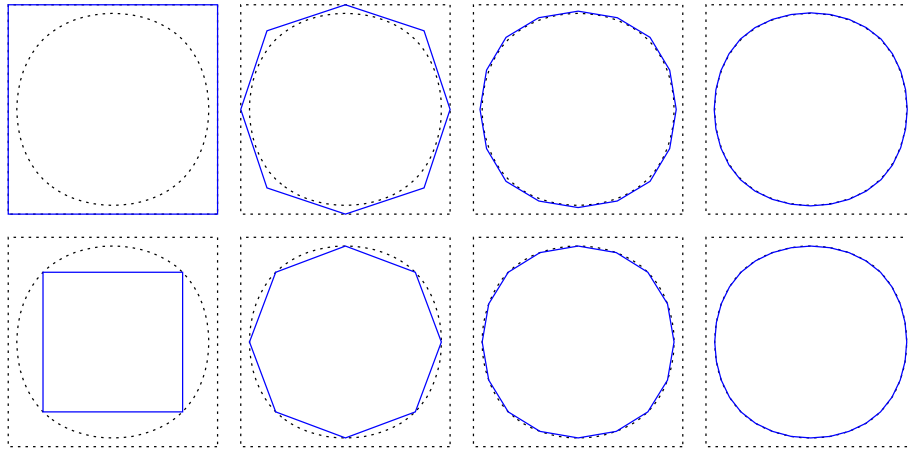


Figure 3.7: Convergence of B-spline curve subdivision. The upper row demonstrates the fast convergence rate: After three steps of subdivision, the square becomes a polygon with 32 line segments that are very close to the limit (dotted curve). The bottom row shows the points from the upper row projected to the limit curve. The great advantage is that a lower resolution can be obtained from a higher resolution curve by simple sub-sampling. The limit curve is not a circle.

As suggested by Fig. 3.6, K^n converges quite rapidly:

$$\begin{aligned}
 K &= \begin{pmatrix} 0.500000 & 0.500000 & 0.000000 \\ 0.125000 & 0.750000 & 0.125000 \\ 0.000000 & 0.500000 & 0.500000 \end{pmatrix} & K^2 &= \begin{pmatrix} 0.312500 & 0.625000 & 0.062500 \\ 0.156250 & 0.687500 & 0.156250 \\ 0.062500 & 0.625000 & 0.312500 \end{pmatrix} \\
 K^3 &= \begin{pmatrix} 0.234375 & 0.656250 & 0.109375 \\ 0.164062 & 0.671875 & 0.164062 \\ 0.109375 & 0.656250 & 0.234375 \end{pmatrix} & K^4 &= \begin{pmatrix} 0.199219 & 0.664062 & 0.136719 \\ 0.166016 & 0.667969 & 0.166016 \\ 0.136719 & 0.664062 & 0.199219 \end{pmatrix} \\
 K^{10} &= \begin{pmatrix} 0.166911 & 0.666667 & 0.166423 \\ 0.166667 & 0.666667 & 0.166667 \\ 0.166423 & 0.666667 & 0.166911 \end{pmatrix} & K^{20} &= \begin{pmatrix} 0.166667 & 0.666667 & 0.166666 \\ 0.166667 & 0.666667 & 0.166667 \\ 0.166666 & 0.666667 & 0.166667 \end{pmatrix}
 \end{aligned}$$

When computing in single precision floating-point (with a 23 bit mantissa), the limit is reached for $n = 23$. The limit matrix K^∞ can be expressed as the dyadic product of $(1, 1, 1)^T$ and the left eigenvector $1/6 \cdot (1, 4, 1)$ of K , which is easy to verify:

$$K \cdot K^\infty = \frac{1}{8} \begin{pmatrix} 4 & 4 & \\ 1 & 6 & 1 \\ & 4 & 4 \end{pmatrix} \cdot \frac{1}{6} \begin{pmatrix} 1 & 4 & 1 \\ 1 & 4 & 1 \\ 1 & 4 & 1 \end{pmatrix} = \frac{1}{8} \cdot \frac{1}{6} \begin{pmatrix} 8 & 32 & 8 \\ 8 & 32 & 8 \\ 8 & 32 & 8 \end{pmatrix} = \frac{1}{6} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} (1, 4, 1) = K^\infty$$

This eigenvector belongs to eigenvalue 1, and it gives the weights of the limit position on the curve, given one vertex and its two neighbours on the control polygon of a B-spline curve. The usefulness of this can be clearly seen in Fig. 3.7. The subdivided control polygon yields exactly the same B-spline curve as the original polygon, shown in dashed. On any such level of subdivision, including level 0, i.e. the original control polygon, the limit position of the polygon vertices can be computed using the *limit weights* $1/6 \cdot (1, 4, 1)$. This has the important consequence that a *polygon refinement* operator can be realized by repeated subdivision followed by application of the limit weights. This last step is referred to as the *projection to the limit position* of a recursively subdivided curve (or surface). The great advantage is then that different parts of the refined polygon may actually be drawn at different refinement levels, and thus at different approximation qualities. Yet still the different parts agree in limit positions, so they can be stitched together seamlessly. This also leads to an efficient implementation, as once a refined polygon is computed, any lower level can be obtained by sub-sampling. This is not the case if only subdivision is applied: The curves in the upper row of Fig. 3.7 (right to left) are not sub-sampled versions of each other, but the curves in the lower row are.

This derivation demonstrates the importance of the eigen analysis of subdivision matrices. As further explained in e.g. the Siggraph Course Notes [ZS99] and the book from Joe Warren [WW02], Eigenanalysis can also be used to derive differential properties such as tangents and curvatures. As it turns out, the tangent vector of a uniform cubic B-spline curve can be computed using the weights $(1, 0, -1)$.

3.1.3 Tensor Product Surfaces

Tensor product surfaces are a quite general way to define a surface from a given curve representation. Recall that in definition 3.2 the B-spline curve took the form $\sum_{i=0}^{k-n-1} p_i N_i^n(\mathbf{t})$. For any parameter $\mathbf{t} \in [t_n, t_{k-n}]$ from the parameter interval, the basis functions sum to one (partition of unity), so all points of the curve are affine combinations of the vertices p_0, \dots, p_{k-n-1} from the control polygon. Furthermore, the coefficients are nonnegative, as $N_i^n(t) \geq 0$, so any point on the curve is actually a *convex combination* of the control vertices. Consequently, the curve lies in the convex hull of the control points. The B-spline functions are not the only example for a basis with these properties: The famous *Bernstein polynomials* $B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}$ also form a nonnegative partition of unity, over the interval $[0, 1]$, and using them for a basis yields the famous Bézier curves. Many other examples for affine basis functions for convex combinations exist [Far02].

The idea of tensor product surfaces is now to consider not only one, but m curves, each with its own set of n control vertices. All m curves are evaluated at the same parameter value v , which yields m points. These points are then used as CVs of another curve, of degree $m-1$, that is evaluated at parameter u .

Definition 3.3 (Tensor product surface) Let $[u_0, u_1]$ and $[v_0, v_1]$ be parameter intervals of two sets of basis functions $A_1(u), \dots, A_m(u)$ and $B_1(v), \dots, B_n(v)$, so that all $A_i(u) \geq 0$, $i = 1, \dots, m$, and $B_j(v) \geq 0$, $j = 1, \dots, n$, and let A and B form partitions of unity $\sum_{i=1}^m A_i(u) = 1$, $\sum_{j=1}^n B_j(v) = 1$ for all parameters from the respective intervals. Furthermore let (p_{ij}) be a grid of $m \times n$ control vertices. The tensor product surface for A , B , and (p_{ij}) is then defined as follows:

$$s(u, v) = \sum_{i=1}^m A_i(u) \left(\sum_{j=1}^n B_j(v) p_{ij} \right) = \sum_{i=1}^m \sum_{j=1}^n A_i(u) B_j(v) \cdot p_{ij} = (A_1(v), \dots, A_m(v)) \begin{pmatrix} p_{11} & \cdots & p_{1n} \\ \vdots & \ddots & \vdots \\ p_{m1} & \cdots & p_{mn} \end{pmatrix} \begin{pmatrix} B_1(u) \\ \vdots \\ B_n(u) \end{pmatrix}$$

The matrix notation $s(u, v) = (B_j(v))^T (p_{ij}) (A_i(u))$ of the surface explains the name “tensor product surface”. Both sets A_i and B_j of basis functions may of course consist of B-spline functions, they may possibly have different degrees and different knot vectors as well. It is even possible to use a B-spline in u and a Bézier curve in v for a mixed representation tensor product patch, or to mix with any other basis function type. The parameter directions are completely independent, important is only the partition of unity to assert that the surface points are convex combinations of the CVs.

In order to derive the Catmull/Clark scheme, the choice is again the uniform knot vector $(-3, -2, -1, 0, 1, 2, 3, 4)$ and the cubic functions from Fig. 3.3. When the uniform cubic basis functions are taken for both directions u and v , the resulting surface can again be parameterized in terms of the monomial basis functions. Recall from the last section 3.1.1 that a B-spline curve can be written in matrix form as $c(t) = P^T M^T T$, with vector of control vertices $P = (p_0, p_1, p_2, p_3)^T$, monomial basis functions $T = (t^3, t^2, t, 1)^T$, and the basis change matrix M . Replacing T by the surface parameters $U = (u^3, u^2, u, 1)$ and $V = (v^3, v^2, v, 1)$ respectively, this can be readily plugged into the tensor product equation:

$$s(u, v) = U^T M P M^T V$$

In order to derive refinement rules for such a tensor product of uniform cubic B-spline functions, literally the same approach as before can be used. The surface is parameterized over $(u, v) \in [0, 1] \times [0, 1]$, and the idea is again to look at the sub-patch $s'(u', v')$ in the “lower left” corner of the parameter space: $(u', v') \in [0, \frac{1}{2}] \times [0, \frac{1}{2}]$. This gives two equations for s' , one obtained by modification of s and another one with its own control vertices, now called Q :

$$\begin{aligned} s'(u', v') &= U^T S M P M^T S^T V && \text{(modified CVs from } s) \\ s'(u', v') &= U^T M Q M^T V && \text{(own grid } Q \text{ of CVs)} \end{aligned}$$

This can be true for all u' and all v' if and only if

$$\begin{aligned} M Q M^T &= S M P M^T S^T \\ \iff Q &= M^{-1} S M \cdot P \cdot M^T S^T M^{-T} \\ &= (M^{-1} S M) \cdot P \cdot (M^{-1} S M)^T \\ &= H P H^T \end{aligned}$$

So the same matrix $H = M^{-1} S M$ as before in the curve case is now multiplied twice to the CV grid, once to the left and once to the right. Intuitively, one might guess that four different refinement rules result from this: In the curve case, applying H once led to two different rules, for vertex and edge points. Applying it twice should lead to vertex/vertex, edge/edge, edge/vertex, and vertex/edge rules. But it turns out that the latter two are identical, so just three different rules result.

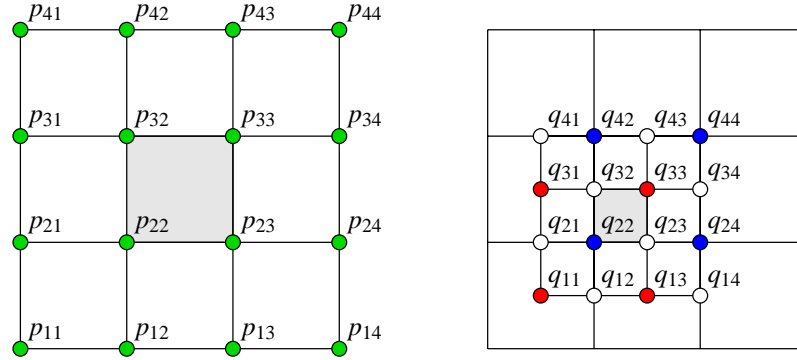


Figure 3.8: Subdivision of the 4×4 control grid of a tensor product surface from uniform cubic B-splines.

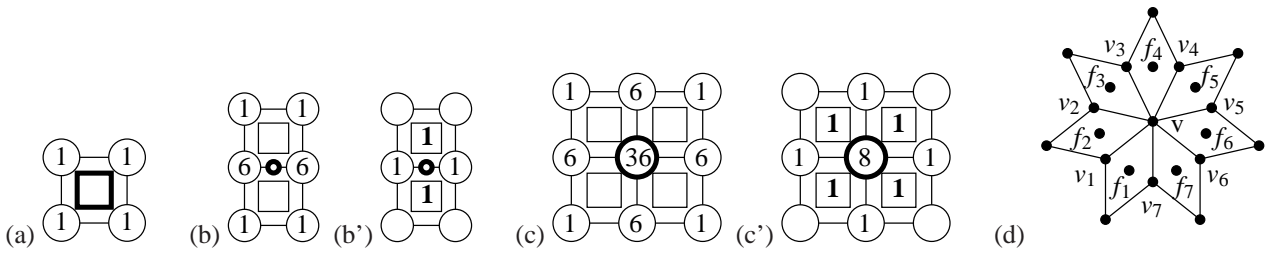


Figure 3.9: Stencils of the Catmull/Clark scheme. (a) face rule: The face point, shown as bold square, of a (regular) face is the average of its (four) vertices. (b), (b') edge rule: An edge point (bold circle) can be expressed as the average of two vertices and two already computed face points. (c), (c') vertex rule, regular case: The vertex point can be expressed as the average of the center vertex and the sum of adjacent vertices and face points. (d): enumeration scheme used in the vertex rule for vertices and already computed face points. In this case all faces are quads, but the rule applies to adjacent faces of any degree. (b') and (c') use the newly computed face points.

To carry out the algebra of the multiplication $Q = HPH^T$ with $H = M^{-1}SM$ is not very handy. What comes out has a remarkable symmetry, though, which is coded in different colors in Fig. 3.8. The refined grid Q is partitioned into three classes: *face points* (red), *edge points* (white), and *vertex points* (blue). For each of these classes, one representative is shown:

$$\begin{aligned}
 \text{face point} \quad \mathbf{q}_{11} &= \frac{1}{4}(p_{11} + p_{12} + p_{21} + p_{22}) \\
 \text{edge point} \quad \mathbf{q}_{12} &= \frac{6}{16}(p_{12} + p_{22}) + \frac{1}{16}(p_{11} + p_{21} + p_{13} + p_{23}) \\
 &= \frac{1}{4}(p_{12} + p_{22}) + \frac{1}{16}(p_{11} + p_{21} + 2p_{12} + 2p_{22} + p_{13} + p_{23}) \\
 &= \frac{1}{4}(p_{12} + p_{22} + \mathbf{q}_{11} + \mathbf{q}_{13}) \\
 \text{vertex point} \quad \mathbf{q}_{22} &= \frac{36}{64}p_{22} + \frac{6}{64}(p_{21} + p_{12} + p_{32} + p_{23}) + \frac{1}{64}(p_{11} + p_{31} + p_{13} + p_{33}) \\
 &= \frac{1}{2}p_{22} + \frac{1}{16}(p_{21} + p_{12} + p_{32} + p_{23}) + \frac{1}{16}(\mathbf{q}_{11} + \mathbf{q}_{31} + \mathbf{q}_{13} + \mathbf{q}_{33})
 \end{aligned}$$

These equations apply in the same way to the other points of the respective classes. To re-group the terms as it is shown has the benefit of reducing the number of operations because the already computed face points can be used (shown in bold). Also note that the operations that are used are numerically extremely stable! Only minimal rounding artefacts are introduced: Divisions by a power of two can be performed *exactly* on binary arithmetic, as they only add to the binary exponent. Furthermore, only additions are used, so that no effects like extinction can occur, which can happen with subtraction.

It is sometimes awkward to match the point indices in the rules with a grid scheme like in Fig. 3.8. A more convenient and intuitive notation, and one that exploits the symmetry of the rules, is *subdivision stencil*. The stencil diagrams show directly how the weights are applied. The entries in the diagrams are only the nominators: as the weights sum to one, the denominator is just the sum of the weights shown in the stencil. Stencils for face, edge, and vertex rules are shown in Fig. 3.9 (a), (b), and (c). The re-grouped versions using the already computed new face points are shown in (b') and (c').

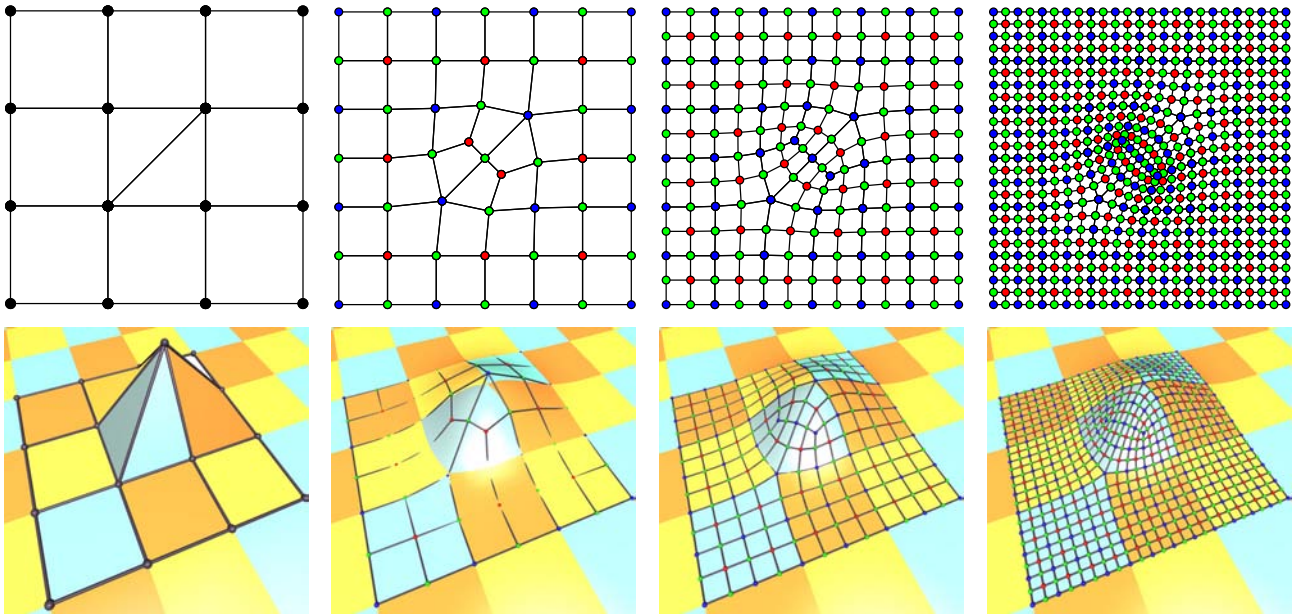


Figure 3.10: Example of recursive Subdivision. Face, edge, and vertex points are red, green, and blue.

3.1.4 Catmull/Clark Surface: Generalization to the Irregular Case

The great achievement of Catmull and Clark was to generalize the rules for the regular case of a 4×4 grid to the irregular case, where face degrees and vertex valences may be different from four. In this case, the edge rule remains unchanged, and the face rule can be generalized in an obvious way. The vertex rule, however, is generalized so that in the regular case, it is identical to the regular vertex rule that is already known.

face rule A *face point* of a face with degree n is the average of the face’s vertices v_1, \dots, v_n , i.e. the face centroid

$$f = \frac{1}{n^2} \sum_{i=1}^n v_i$$

edge rule An *edge point* is the average of the two endpoints of the edge and the two face points of the incident faces.

$$e = \frac{1}{4} (v_0 + v_1 + f_0 + f_1)$$

vertex rule A *vertex point* of a vertex with valence n is computed from v itself, from the adjacent neighbour vertices v_1, \dots, v_n , and from the newly computed face points f_1, \dots, f_n of the incident faces (indexing scheme in Fig. 3.9 (d)):

$$v' = \frac{n-2}{n} v + \frac{1}{n^2} \sum_{i=1}^n v_i + \frac{1}{n^2} \sum_{i=1}^n f_i$$

mesh connectivity rule A refined mesh is formed by connecting all newly computed face and vertex points to the edge points around them.

Why are irregular control meshes so important? The generalized rules remedy an important limitation of B-spline surfaces, namely that only regular grids of control vertices can be used. This is of great practical importance. To form a shape with a B-spline surface, an artist can move the CVs from the regular $m \times n$ -grid freely around in space. Now suppose the artist wants to apply a bump to a particular place on the surface, but it happens that there is no CV nearby, i.e., no appropriate degree of freedom (DOF) is available. To *insert* a new local DOF is possible, but this changes the whole grid: A whole new row and a new column of CVs have to be inserted using *knot insertion* in both directions. And all this effort only to let the artist move the new CV at the intersection of the new row and column! – This is the reason why the number of CVs always tends to increase heavily with all kinds of tensor product surfaces.

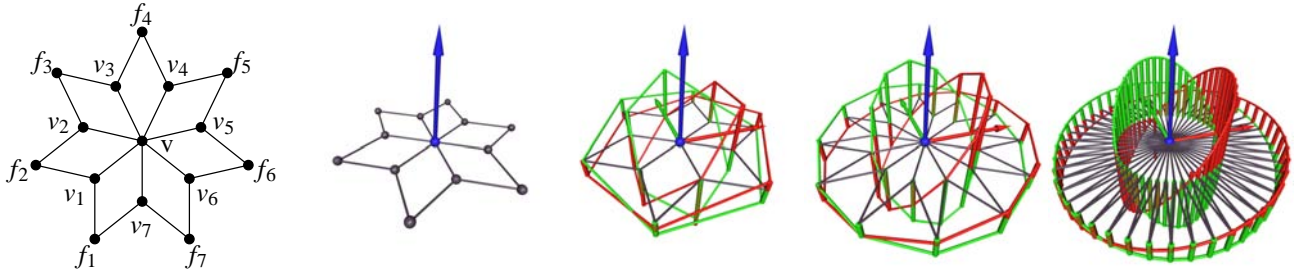


Figure 3.11: Stencil for limit position and tangents. (a): Valence 7 example. Note that in this enumeration scheme, the f_i do not refer to the face centroids but to the opposite quad vertices. So the limit rules only apply when all incident faces are quads, which is the case right after the first subdivision. (b-e): Visualization of the weights for limit tangents with increasing vertex valence. Sine and cosine have the suitable property that they are identical but shifted by 90 degrees.

With subdivision surfaces an artist can modify the control mesh *locally* if the surface needs more detail only in this place. In other words, new degrees of freedom can be introduced only where needed, without affecting the rest of the surface. The following arguments from [HKD93] nicely sum up the properties of Catmull/Clark surfaces:

- This set of rules can be applied to any closed valid 2-manifold mesh, the *control mesh*.
- The faces can have any degree, vertices may be of any valence.
- After the first subdivision all faces are quadrilaterals.
- Around regular vertices (valence 4) the surface is a bicubic B-spline surface.
- The limit surface is curvature continuous except at irregular vertices.
- The number of irregular vertices remains constant after the first subdivision.

Due to the fact that only faces from the control mesh may have a degree unequal to four, it is convenient to introduce some notation to distinguish between the control mesh and faces produced by subdivision. In the following, the control mesh will also be referred to as the *base mesh*, and its respective entities are *base faces*, *base edges*, and *base vertices*. The terms ‘control mesh’ and ‘base mesh’ denote the same thing; the difference lies in the connotation. While the first emphasizes the artistic or design aspect, the latter comes from the multi-resolution community and is used when reasoning, e.g., about recursive refinement.

It is important to note that the generalized face and vertex rules share one important property with the original B-spline refinement rules: They are linear in the positions of vertices and faces. As a particular consequence, a subdivision step can be formally described as a linear mapping, i.e., a matrix: The *subdivision matrix*.

After the first subdivision step all faces are quads. Any irregularity can then only come from the vertices. So the difference between the regular setting, as in Fig. 3.8, and the irregular case is that the vertices $p_{22}, p_{23}, p_{32}, p_{33}$ of the center quad can have any valence. It is therefore more convenient to consider only the situation around a single vertex, as it is done in Fig. 3.9 (d), or, more commonly, as in Fig. 3.11. In this setting, no single subdivision matrix exists, but there is one matrix \mathbf{S}_n for each vertex valence. The valence n subdivision matrix \mathbf{S}_n takes the positions of the n direct and n indirect neighbours v_i and f_i around a valence n vertex v , arranged as column vector, to the next refinement level:

$$(v', v'_1, f'_1, \dots, v'_n, f'_n)^T = \mathbf{S}_n (v, v_1, f_1, \dots, v_n, f_n)^T$$

3.1.5 Rules for the Limit Position and Surface Tangents

The subdivision rules of Catmull/Clark can be recursively applied to obtain a finer and finer mesh, which in the limit converges to the smooth surface itself. But in every step, each quadrangle face is split into four smaller quadrangles. So the number of faces grows exponentially, as a mesh with n quadrangles after the first subdivision will have $4^k \cdot n$ quadrangles after k more subdivision steps.

Just like in the curve case, this process will eventually converge to a limit surface. Every base vertex corresponds to a unique vertex from the k times refined mesh. Such a vertex is called a *child on the k -th subdivision level*. So in particular, every base vertex has a unique position on the limit surface, which is called its *projection on the limit surface*.

Following the same approach as in the curve case, rules for the limit position can be obtained by analysis of the subdivision matrix. As for curves, this matrix comes from looking at the situation around a single vertex. But this needs to be done for each valence. Fortunately, a closed solution exists. The following remarkable rules can be found in Appendix A of the paper from Halstead et al. [HKD93].

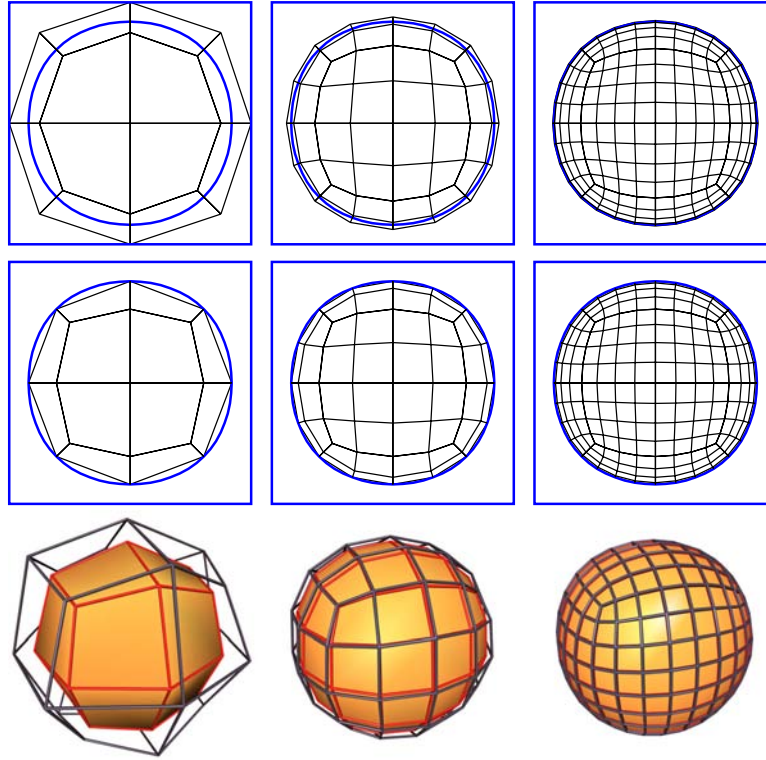


Figure 3.12: Comparison of subdivision alone to subdivision with subsequent projection to the limit surface. The blue square is the orthographic projection of the control mesh, a cube. The blue curve is the silhouette projection of the limit surface. This curve is identical to the curve in Fig. 3.7. Note that the corners of the cube have valence 3, so on every subdivision level there are just eight irregular vertices, with valence 3, the children of the corners. Note that as in the curve case, subdivision approaches the limit surface by shrinking, while projecting the subdivided points to the limit surface yields a denser and denser sampling without shrinking effects. This sampling scheme also has the sub-sampling property, which is important for adaptive evaluation. The difference between subdivided grid (black) and limit surface grid (red) is also shown in the bottom row. Bottom right image: After 3 levels of subdivision the difference is small but noticeable, as a little bit red can still be seen.

$$\begin{aligned}
 \text{limit vertex rule} \quad v^\infty &= \frac{n}{n+5} v + \frac{4}{n(n+5)} \sum_{i=1}^n e_i + \frac{1}{n(n+5)} \sum_{i=1}^n f_i \\
 \text{limit tangent rule} \quad t_x^\infty &= \sum_{i=1}^n A_n \cos\left(\frac{2\pi i}{n}\right) e_i + \sum_{i=1}^n \left(\cos\left(\frac{2\pi i}{n}\right) + \cos\left(\frac{2\pi(i+1)}{n}\right) \right) f_i \\
 t_y^\infty &= \sum_{i=1}^n A_n \sin\left(\frac{2\pi i}{n}\right) e_i + \sum_{i=1}^n \left(\sin\left(\frac{2\pi i}{n}\right) + \sin\left(\frac{2\pi(i+1)}{n}\right) \right) f_i \\
 \text{where} \quad A_n &= 1 + \cos\left(\frac{2\pi}{n}\right) + \cos\left(\frac{\pi}{n}\right) \sqrt{18 + 2 \cos\left(\frac{2\pi}{n}\right)} \\
 \text{limit normal rule} \quad n^\infty &= t_x^\infty \times t_y^\infty
 \end{aligned}$$

These rules are somewhat mystic, and far from obvious. Halstead et al. do not provide much insight on how they obtained them. A computer algebra system employed complained about Eigenvalues being complex. Jos Stam, who derives part of some limit rules in his ‘evaluation’ papers [Sta98, Sta99], only mentions a technique to ‘rotate complex entries away’. – Some insight on the question why so many sine/cosine terms appear is at least provided by Fig. 3.11 (b-e).

The limit position of a base vertex is unique, so the limit rules yield identical results when applied to a base vertex, or to any of its direct children. This is important for adaptive surface refinement, because it means that, as in the curve case (Fig. 3.7), a lower resolution can be obtained from a higher refinement level by simple sub-sampling also for surfaces (Fig. 3.12). Note that the fast convergence rate from curves carries over to surfaces.

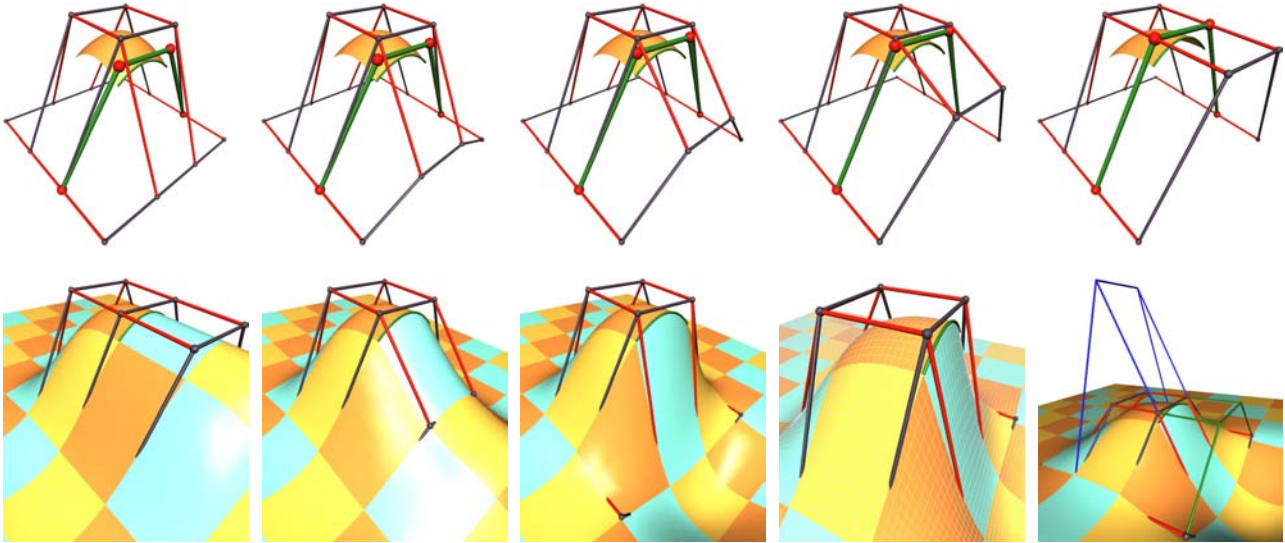


Figure 3.13: Crease curve design.

Upper row: A 4×4 control mesh of a bicubic B-spline patch. One border of the patch is marked in green, this is a B-spline curve with four control points, marked with green sticks: Its CVs are the four big red points, which are the result of the evaluation of the four row curves (thin red sticks). When the left red segments are identical to the segments before (collinear, same length), the green control polygon of the border curve coincides with the 3rd column of the 4×4 grid (top row, last image). This is exactly the configuration used with crease curves.

Bottom row: The 3rd column is now made a crease. The cross-border tangents are decoupled, but the surface remains connected (C^0 continuity). Bottom row, last image: The virtual last rows of the control mesh of the surface to the left (continued in green) and to the right (continued in blue) of the crease.

3.1.6 Borders and Creases

The rules presented so far work well for closed manifold meshes. But for control meshes containing borders the rule set must be extended. This extended rule set can also be used to introduce a new feature, namely *crease curves*.

As was demonstrated in the previous sections, Catmull/Clark surfaces are a generalization of B-spline tensor product surfaces. The construction of these tensor product surfaces is quite elegant, because they basically “lift” one-dimensional curves to a two-dimensional surface. Thanks to the *partition of unity* property of the underlying curves, the four borders of a tensor product surface patch are again curves of the same type, as one parameter is fixed:

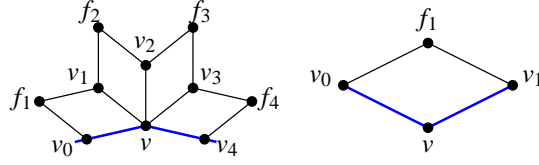
$$\begin{aligned} s(u, 0) &= \sum_{i=1}^m A_i(u) \left(\sum_{j=1}^n B_j(0) p_{ij} \right) & s(0, v) &= \sum_{j=1}^n B_j(v) \left(\sum_{i=1}^m A_i(0) p_{ij} \right) \\ s(u, 1) &= \sum_{i=1}^m A_i(u) \left(\sum_{j=1}^n B_j(1) p_{ij} \right) & s(1, v) &= \sum_{j=1}^n B_j(v) \left(\sum_{i=1}^m A_i(1) p_{ij} \right) \end{aligned}$$

When A and B are B-spline basis functions the borders are ordinary B-spline curves. To obtain the CVs of the border curve just one surface parameter is fixed. In case of a regular bi-cubic B-spline patch with 4×4 CVs, the four CVs of the $u = 1$ border curve are obtained from evaluation of the four row-wise B-spline curves. Their control polygons are shown in red in Fig. 3.13. This behaviour has two undesirable effects for meshes with border:

1. The surface depends on CVs that have no corresponding point on the free-form surface.
2. It is not obvious how to stitch another base mesh to a given border so that there is no gap between the surfaces.

The first objection is of a somewhat aesthetic nature. But the second problem is serious, because for a neighbouring patch to fit the expressions in the brackets need to be identical. An example is shown in the top left image in Fig. 3.13: The red balls form the control polygon (green) of the border curve (also green). Note in the bottom row that the left part of the surface does not change its shape no matter how the surface on the other side of the crease is changed. The patch is still identical to the patch shown in the last image of the upper row. Both sides of the surface coincide only in the crease curve, which is the B-spline curve defined by the base mesh CVs on the crease. This polygon needs to be matched by the respective polygon of a neighbouring patch to close the gap.

But there is a simple solution to this problem: Let the last row be only “virtual”, and compute it as the continuation of the center segments. This situation is shown in the top right image (curves with red segments), and its use for designing a crease in a continuous surface is shown in the bottom right image. Recall that for B-spline curves, the limit position

Figure 3.14: Stencil for crease limit position and tangents, with $k = 4$.

of a control polygon vertex can be computed using the weights $(1, 4, 1)/6$ from its two neighbours on the curve. Let the polygon be (p_0, p_1, p_2, p_3) and p_2 be the point in question. So when is p_2 identical to its own limit position?

$$p_2 = p_2^\infty = \frac{1}{6}p_1 + \frac{4}{6}p_2 + \frac{1}{6}p_3 = p_2 + \frac{1}{6}(p_1 + p_3 - 2p_2) \iff p_1 - p_2 = p_2 - p_3$$

So this is the case when the last two segments are identical, i.e., when they are collinear and have the same length. Practically, to use it with surface subdivision, a crease is defined by a sequence of edges in the control mesh, and the respective crease curve is obtained by regarding this sequence as the control polygon of an ordinary (uniform) cubic B-spline curve. The subdivision surfaces next to the crease simply act as if the control mesh was continued across the crease in the fashion shown in the bottom right image from Fig. 3.13. Note that this also solves the first problem: The supplemental rows are only virtual, so the control mesh is still a closed manifold mesh.

The definition of a crease as a B-spline curve also works well together with subdivision. Only two new rules are needed, one for refining existing vertices, and one for generating new vertices on a crease edge. Both rules are directly taken from curve subdivision (cf. sec. 3.1.2).

crease edge rule A new *crease edge point* is obtained as the average of the two endpoints of the edge:

$$e_{\text{crease}} = \frac{1}{2}(v_0 + v_1)$$

crease vertex rule A *crease vertex* is a vertex v incident to exactly two edges tagged as creases. In this case the vertex point is obtained from the neighbouring vertices $v_{\text{prev}}, v_{\text{next}}$ on the crease just as in the curve case. The limit position of v is also on the curve and does not depend on the surface CVs.

$$\begin{aligned} v' &= \frac{1}{8}(v_{\text{prev}} + 6v + v_{\text{next}}) \\ v^\infty &= \frac{1}{6}(v_{\text{prev}} + 4v + v_{\text{next}}) \end{aligned}$$

crease normal rule A crease vertex has two normal vectors attached to it, for the surface on either side of the crease. The tangent vector along the crease is very simple, but the cross-crease surface tangent is more complicated. Let $v_0 = v_{\text{prev}}$ and $v_k = v_{\text{next}}$ be the neighbouring crease vertices, and $f_0, v_1, f_1, \dots, v_{k-1}, f_{k-1}$ the face points and neighbour vertices to the left of the crease v_0, v, v_k (see Fig. 3.14)

$$\begin{aligned} t_x^\infty &= v_k - v_0 \\ t_y^\infty &= \alpha v + \beta_0 v_0 + \sum_{i=1}^{k-1} \beta_i v_i + \sum_{i=1}^{k-1} \gamma_i f_i + \beta_k v_k \\ n^\infty &= t_x^\infty \times t_y^\infty \end{aligned}$$

where for $k = 1$: $\alpha = -4$ $\beta_0 = \beta_1 = 1$, $\gamma_0 = 2$,

and for $k > 1$: for $i = 1 \dots k-1$, and with $R_k = \frac{1 + \cos(\frac{\pi}{k})}{k \sin(\frac{\pi}{k}) (3 + \cos(\frac{\pi}{k}))}$,

$$\begin{aligned} \beta_0 = \beta_k &= -R_k(1 + 2\cos(\frac{\pi}{k})) & \beta_i &= \frac{4 \sin(i \frac{\pi}{k})}{k (3 + \cos(\frac{\pi}{k}))} \\ \gamma_i &= \frac{\sin(i \frac{\pi}{k}) + \sin((i+1) \frac{\pi}{k})}{k (3 + \cos(\frac{\pi}{k}))} & \alpha &= 4R_k(-1 + \cos(\frac{\pi}{k})) \end{aligned}$$

The rule for the cross-crease tangent t_y^∞ appears in the appendix of a paper from Biermann et al. [BLZ00]. For the case $k = 2$, i.e., when a crease goes along two edges of the same face of the base mesh, they appear to propose the weights $\alpha = 6, \beta_0 = \beta_1 = -3, \gamma_0 = 0$. This case is shown in Fig. 3.16 in the second image, where the crease follows the face borders in four vertices, while the two vertices in the middle have on both sides $k = 2$. The proposed weights pose difficulties when $k = 1$ and $v_{\text{prev}}, v, v_{\text{prev}}$ are collinear, as in this case the two tangents are collinear as well, so that the normal has length zero.

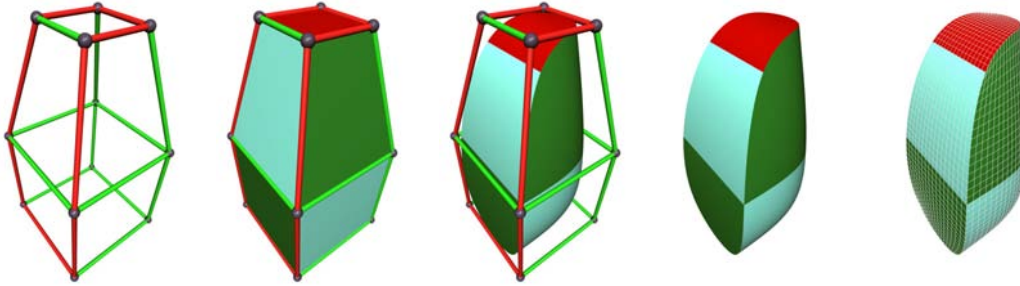


Figure 3.15: Simple example of an object with a crease. Examine the closer top face vertex.

Corners and dart vertices. The motivation of creases was to find a simple rule for efficiently treating meshes that are not closed but have a border. Of course, any border is topologically equivalent to a closed curve. With the crease approach, B-spline curve subdivision is plugged in to Catmull/Clark subdivision, and two open surfaces with the same border polygons will seamlessly meet along the curve.

But creases can also very well be used with closed manifold meshes. In this setting, each edge of the control mesh carries a boolean *sharpness flag* that determines whether the respective edge is *smooth* or *sharp*, i.e., a crease edge. With the ordinary smooth edges, the unmodified rules are used. When edges can be set to sharp without any restrictions, two more cases must be handled in addition to crease vertices: Vertices that are incident to only one sharp edge, the *dart vertices*, and vertices with more than two sharp edges, the *corners*.

dart vertex rule A *dart vertex* is a vertex incident to one sharp edge and an arbitrary number of smooth edges. Its vertex point is computed using the standard vertex rule. The standard vertex limit rules do not apply, though. One workaround is to apply them only after a number of subdivision steps.

corner vertex rule A *corner vertex* is a vertex v incident to more than two sharp edges. It remains fixed at the same position across all subdivision levels. A corner with m sharp edges has m normal vectors, one for each wedge. A *wedge* is formed by two consecutive sharp edges and the smooth edges between them, as found when traveling around the vertex. The normal vector of a wedge is the cross product of the sharp edges' directions:

$$n_v^\infty = \frac{(v_0 - v) \times (v_k - v)}{\| (v_0 - v) \times (v_k - v) \|}$$

The indices are the same as for crease vertices in Fig. 3.14. Visually more pleasing results however can be obtained by averaging all face normals of a wedge:

$$n_v^\infty = \frac{s}{\|s\|} \quad \text{where} \quad s = \sum_{i=1}^k \frac{(v_{i-1} - v) \times (v_i - v)}{\| (v_{i-1} - v) \times (v_i - v) \|}$$

Irrespective of the orientation of the smooth edges in between a wedge's two sharp edges, the surface at a corner locally approaches the plane spanned by the sharp edges. This is a delicate subject, though: Close to the corner the surface may exhibit a very high curvature, so that the plane is a good surface approximation only in a small neighbourhood around the vertex. This occurs for example when the smooth edges in a wedge deviate much from the plane. The problem gets even worse if the opening angle of a wedge is more than 180 degrees, which is not an uncommon situation. This led to the above heuristic. The subject was treated in detail, though, in the 'normal control' paper by Biermann, Levin, and Zorin [BLZ00]. They propose a slightly modified rule set that takes the wedge opening angle into account. Unfortunately it is patented (oral communication), so it was not included in any of our group's software.

There is also a problem with the limit weights of dart vertices: They are definitely not the same weights as for smooth vertices. The reason is that the limit weights are derived from the eigenanalysis of the subdivision matrix – and this matrix is different for dart vertices. It differs only in one row, though, namely the row to compute the edge point of the crease. But this difference leads to a different limit behaviour: In a smooth unit cube (edge length 1 unit) with one sharp edge, the limit position of the two dart vertices differs by 0.0768499 units or about 7.7%, depending on whether the smooth limit rules are applied to the 1 times or the 4 times subdivided base mesh. The respective normals differ by more than 11 degrees. This workaround works well in practice. A thorough solution of the problem would involve the analysis of the eigenstructure of the subdivision matrix for dart vertices – which unfortunately could not be found in the literature.

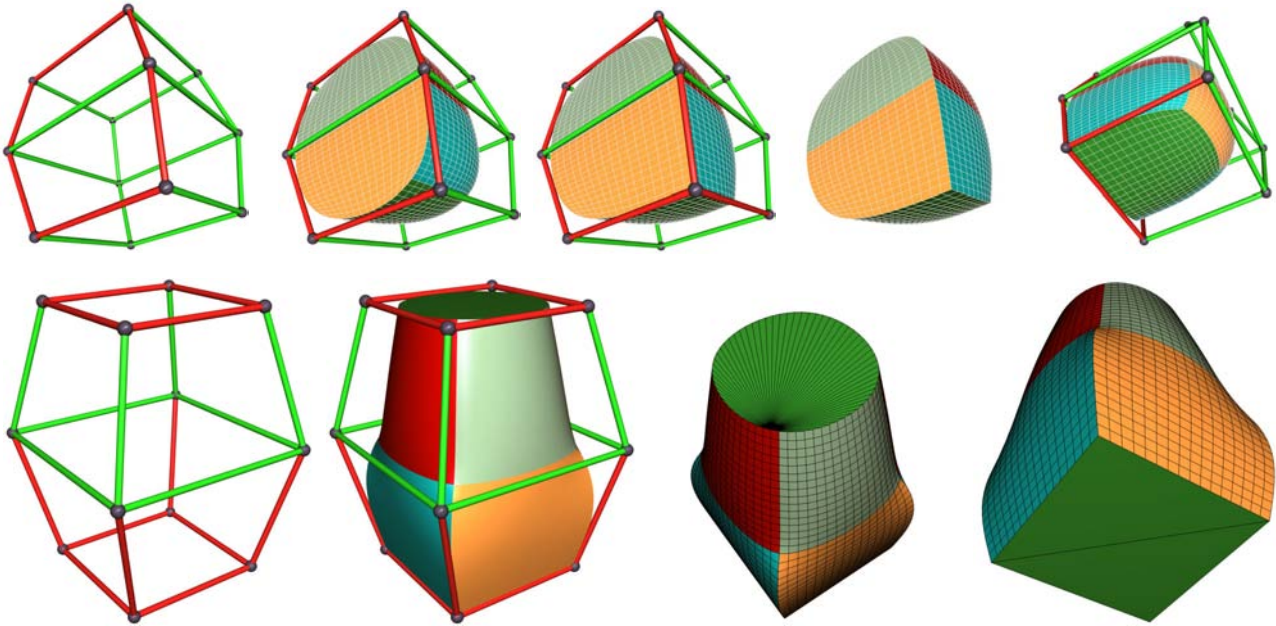


Figure 3.16: Examples of different vertex and face types. Top row: Crease edges do not have to form closed loops. When a crease vertex gets an additional sharp edge, it turns into a corner vertex. A vertex may also be incident to a single sharp edge, which turns it into a dart vertex: The sharp crease ends within a smooth surface. Bottom row: The face classification follows the vertex classification. Both the top and bottom faces have only sharp edges. But the vertices of the bottom face are also corners, as they are incident to three sharp edges. All edges are therefore straight line segments, and the polygonal quad face can simply be rendered using two triangles. The top face is a sharp face, and the B-spline curves must be sampled prior to triangulation.

3.1.7 Artist's Delight: Summary of Vertex and Face Classifications

Smooth vertex	0 sharp edges,	control point of the freeform surface
Dart vertex	1 sharp edge,	endpoint of a crease curve
Crease vertex	2 sharp edges,	control point of a crease curve
Corner vertex	≥ 3 sharp edges,	vertex remains fix across all refinement levels

Table 3.1: Vertex classification according to the number of incident sharp edges

Smooth face	a face with at least one smooth edge, the face has no rings
Sharp face	all edges are sharp, at least one crease vertex
Polygonal face	all edges are sharp, and all vertices are corners

Table 3.2: Face classification according to the vertex classification and edge types.

The possibility to freely choose which edges are sharp and which are smooth results in a shape representation that offers the utmost freedom and expressiveness to creative designers and artists. There is already a leap in productivity by using subdivision surface: Instead of having to stitch together a complicated free-form object from isolated tensor product patches, there is only one single control mesh for the whole shape. The control mesh can have any connectivity, and the artist may modify the control mesh anywhere to add detail to the surface.

For the resulting surface to behave as intuitively as possible, there is the two sets of simple classification rules above. The classification proceeds from vertices to faces: The face classification depends on whether the face has any smooth edge, and if not, on whether all edges are straight line segments or not. Concerning the face classification, two cases have not yet been discussed, which actually help to bridge the gap between polygonal and free-form surfaces: Polygonal and sharp faces. Faces of both types have exclusively sharp edges, and they will be treated in detail in section 4.3 in the next chapter.

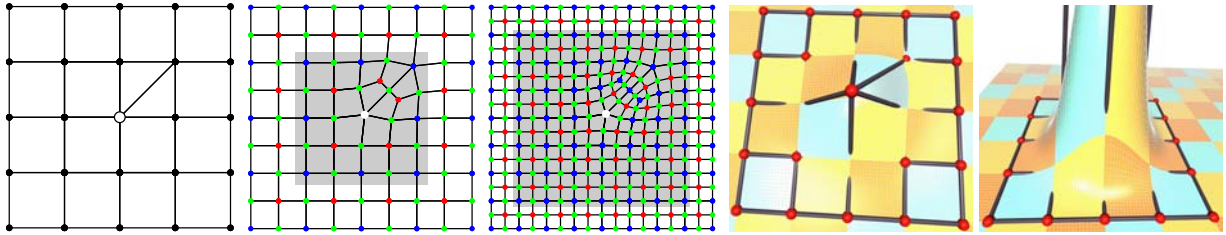


Figure 3.17: Influence of a single base vertex. The position of the white control vertex (first image) directly influences the red face centroid of the first subdivision. The centroids go into the surrounding edge points (green), and finally into the vertex points (blue). The darkened region denotes the region of influence in the first and second subdivision. Note that on any level, all vertices except the two outer rings are influenced. Images 4 and 5: The CV is lifted first a bit, then a lot. This demonstrates which portion of the surface is lifted together with it, and that the border of the 2-neighbourhood remains in place.

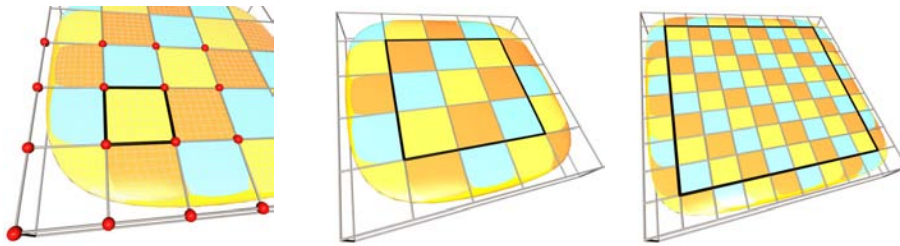


Figure 3.18: Identity of the base faces and subdivision surface: For all configurations of 4×4 control vertices in an equally spaced grid, the center face of the control mesh is identical to the corresponding subdivision surface.

3.2 Practical Experiences with Subdivision Surfaces

This section shall give a somewhat informal, and certainly incomplete, overview about some *qualitative* properties of subdivision surfaces. It shows which problems may occur when they are employed for 3D modeling in practice. The purpose is to better understand the implications of the formal rules and equations from the previous sections; some of these implications are less obvious. The purpose of this exercise is to gain a better intuition for the tasks subdivision is very good at, and which mesh configurations inevitably lead to problems – and why.

This research is informal when it comes to comparisons between the control mesh and the subdivision surface: Strictly spoken, the control mesh does not define a surface. Of course it contains topological faces, but as there is no constraint on vertex positions, in general no polygonal surface can be attached to the faces. A face is only defined by its boundary which is an arbitrary, piecewise linear, closed path in 3D. The subdivision surface, on the other hand, exists even for pathological control meshes, with self-intersections and all other geometrical pitfalls (see Fig. 2.28): It is defined as the limit of a bounded infinite refinement process that provably converges.

Although the control mesh and the subdivision surface can not really be compared, it is instructive to look at how the subdivision behaves in relation to the control mesh. This uses the informal notion of a *control mesh surface*. One can also speak of the subdivision surface corresponding to a base face: One base face together with its 1-neighbourhood is enough to start the refinement process. Such a portion of the subdivision surface is also called a *patch*. This section also demonstrates subdivision on some real models, taking the promise literal that any manifold mesh can be used as control mesh for subdivision.

3.2.1 Radius of Influence of a Vertex and a Face

A Catmull/Clark surface is created basically by “sliding” the 4×4 control mesh of a uniform cubic B-spline surface over a given base mesh. But which parts of a surface are influenced by one specific vertex v ? The vertex coordinates are propagated by the subdivision rules to the surrounding surface. So one needs to look at how the rules proceed:

- v influences the face centroids f_i of all surrounding faces
- The f_i are used for computing all the edge and vertex points of these faces.
- In each round of subdivision, these entities spread the influence of v one step further in the neighbourhood
- But with each step, the faces are subdivided once – and the influence spreads half as wide.

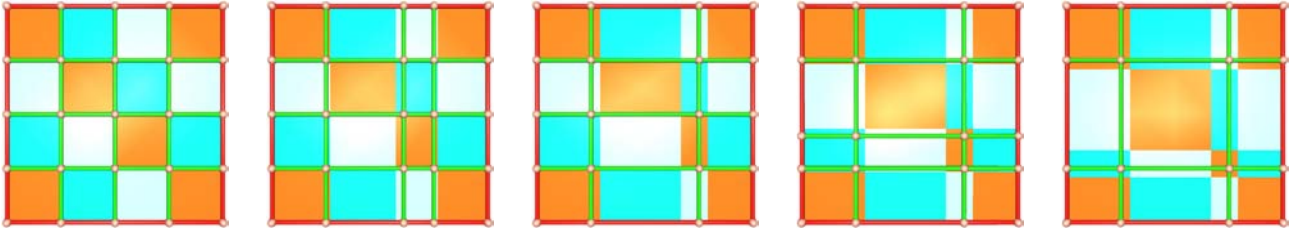


Figure 3.19: An example of surface shift: Faces with zero area can very well have a subdivision surface with non-zero area. Subdivision implies averaging, so large faces get smaller while small faces get larger – always in relation to the size of the direct neighbours. This is demonstrated here by collapsing two rows and two columns of a regular control mesh. The face where the two rows and columns cross has collapsed into a point with no area at all. The corresponding subdivision surface is shifted a little towards the bigger top left neighbour face.

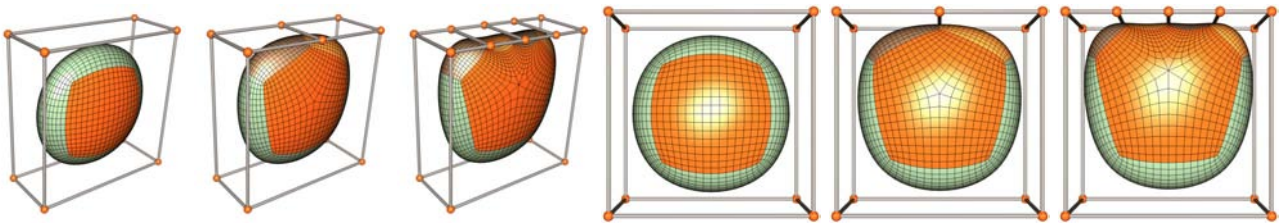


Figure 3.20: A shift of the face centroid leads to surface shift: The centroid is the average of the vertices, so it goes into the direction where most of the vertices are – and it drags the rest of the surface with it.

It is also instructive to consider a regular grid of unit quadrangles so that the subdivision surface is reduced to the regular B-spline case. The radius k of influence increases with each subdivision step according to the sequence $1, 1/2, 1/4, 1/8, \dots, 1/2^n$. The sum $\sum 2^{-n}$ approaches 2, which is v 's radius of influence. So v influences the surface of $4 \times 4 = 16$ faces around it. See Fig. 3.17 for a similar example. As far as limit positions of base vertices are concerned, however, a vertex contributes only to the limit positions of its direct neighbours.

So note: When a vertex is moved, the whole subdivision surface in its “open” topological 2-neighbourhood is changed and must eventually be recomputed. This may lead to unexpected behaviour when a vertex has one very long edge – or when a vertex's neighbour vertex has a long edge.

3.2.2 Smoothing, Averaging, and Surface Shift

Many of the figures so far were based on the regular setting, i.e., a grid of regular planar quadrangles in the control mesh. In this special case the B-spline surface is exactly identical to the quad itself. This holds also locally: Whenever there is a configuration of 16 vertices like in Fig. 3.18 (c), the center quad will be exactly interpolated by the subdivision surface. But in general, the control mesh is not planar, faces have different sizes, and the face degrees and vertices valences are different from four. In these cases the subdivision surface only roughly follows the control mesh. Both surfaces not only differ in normal direction, but their difference has also a tangential component. This effect is called *surface shift*, and it sometimes leads to results that are counter-intuitive at first sight, so it is important to understand why and how the surface is shifted.

The vertices of the subdivided grid are not only affine combinations but also *convex combinations* of the base vertices: All subdivision rules are of the form $v^{\text{new}} = \sum \alpha_i v_i^{\text{old}}$, with $\sum \alpha_i = 1$ and for all i , $\alpha_i \geq 0$. So this process is basically some averaging. But averaging means that differences are reduced:

- sharp peaks and sharp edges are rounded,
- subdivided quads have a more regular quadrangular shape, i.e., they are approaching a square,
- the angles between the edges around a vertex become more regular, i.e., they approach $2\pi/n$ for valence n , and
- *surface shift*: large faces become smaller, small faces become larger

One effect of averaging is that even base faces with no area at all can have a non-vanishing subdivision surface, as demonstrated in Fig. 3.19: The surface is ‘blown up’ on the expense of the subdivision surface from neighbour faces. To make a subdivision surface vanish, all CVs in the 1-neighbourhood of the base face must be degenerate, i.e., must collapse to the same point, or lie on a line.

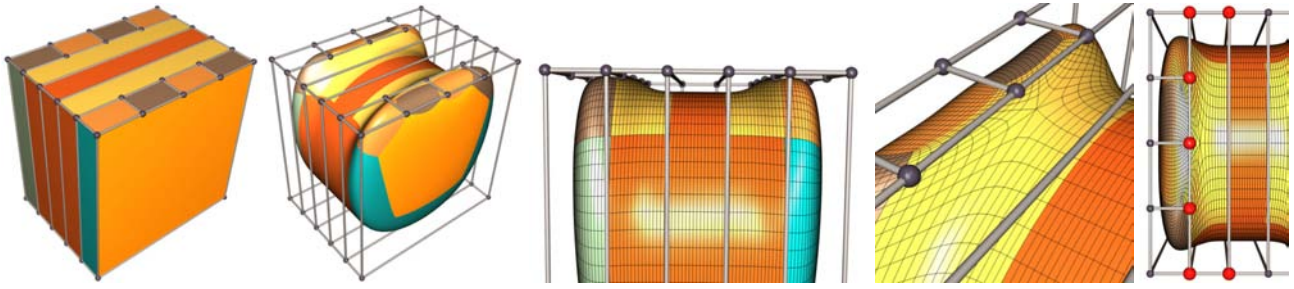


Figure 3.21: Many vertices attract the surface. In particular they attract the face point.

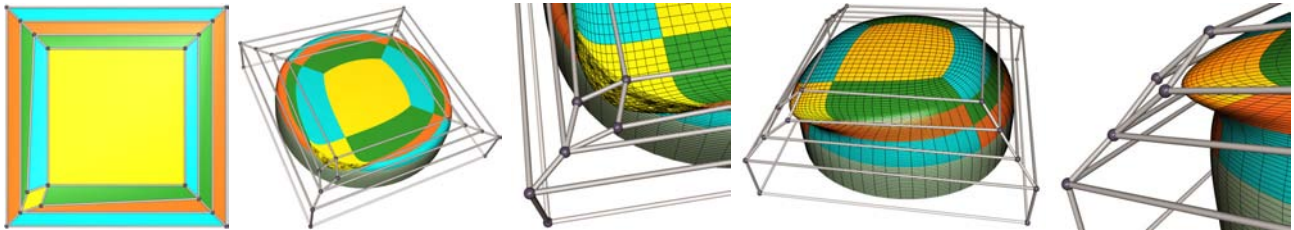


Figure 3.22: Foldover example. Small faces attract the surface.

Surface shift and fold-overs. Subdivision starts by computing the centroid of each face from the base mesh: The centroid is the average of all of its vertices, which are a set of distinguished points on the face border. It is *not*, however, the center of mass of the face as a two-dimensional set of points in \mathbb{R}^3 . Averaging is rather a majority decision, so the centroid is attracted by the location where most of the vertices are. This is demonstrated in Fig. 3.20, where the centroid is shifted to the top of the front quad. With further refinement, the whole surface basically follows the shifted centroid.

Another way to look at such conglomerations of vertices is to relate the sizes of neighbouring faces: Vertices that are close together attract the subdivision surface more, but they also make for smaller faces. As a rule of thumb, it can therefore be stated that small faces attract the subdivision surface stronger than large faces.

A prototypical situation is shown in Fig. 3.21: A stack of quadrangles is refined on bottom and top just as in the previous example. As a result, the two refined quads attract the subdivision surface more than their neighbours. The subdivision surface is therefore on both ends closer to the base mesh than in the middle (Fig. 3.21 (c)). Fig. 3.21 (d) shows how close the centroid is to one vertex. The reason is that the respective face is just a simple 7-gon (3.21 (e)) where most of the vertices are to the left; note that the centroid also has valence 7, and its edges basically point to the midpoints between the face's vertices (actually to the edge points between them).

This behaviour can sometimes lead to unforeseen *fold-overs* of the surface. The object in Fig. 3.22 (a,b) has basically the same topology as the one in the previous example (Fig. 3.21): A stack of quads, but this time the quads remain in the same plane, but are shrunk. The difference to the previous example is the modification, namely a small quad in the lower left corner of the surface. This quad is much smaller than the surrounding faces. But the subdivision surface is completely determined by the 1-neighbourhood of a surface, and so the small quad attracts the surface. As the quad is in the same plane as its neighbours, the result is a fold-over (Fig. 3.22 (c)).

The situation becomes clearer when the stacked (extruded) faces are put on different heights, as in 3.22 (d, e): The smaller quad makes that the subdivision forms a clear 'nose', which becomes the fold-over when the control mesh is flattened.

3.2.3 Linearity and Ripples

Subdivision is a linear process. The refinement rules are such that face, edge, and vertex points on the next level all linearly depend on the positions from the level immediately before. It is quite remarkable, actually – although plausible from the linear algebra point of view – that the result of infinitely repeating a linear mapping can itself also be obtained through a linear mapping: The limit rules are linear in the CVs.

A practical consequence of this linearity is shown in Fig. 3.23: When an artist is moving a vertex, the subdivision surface follows in a completely predictable manner: Twice the displacement of the CV gives twice the displacement of the surface, yet the absolute distance of the surface displacements are smaller, of course.

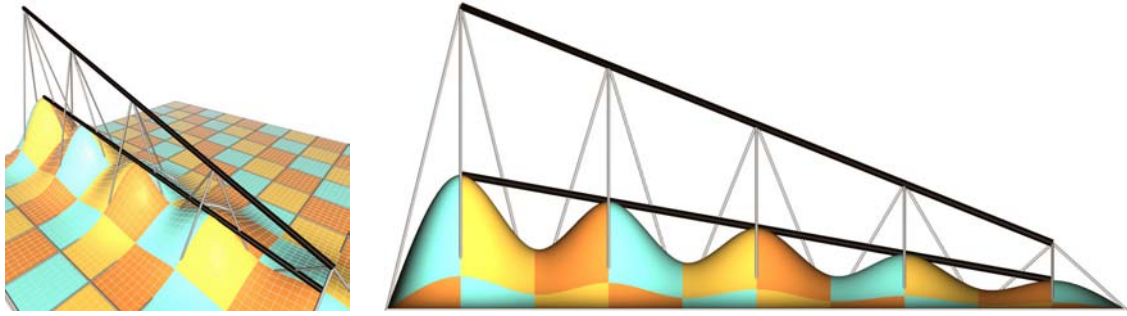


Figure 3.23: Linear dependency between the subdivision surface and the displacements of the CVs. The limit position of a vertex depends only on the CV itself and its direct neighbours. When modifying the surface, an artist can influence the surface in a very predictable way when keeping an eye on the CV's limit positions.

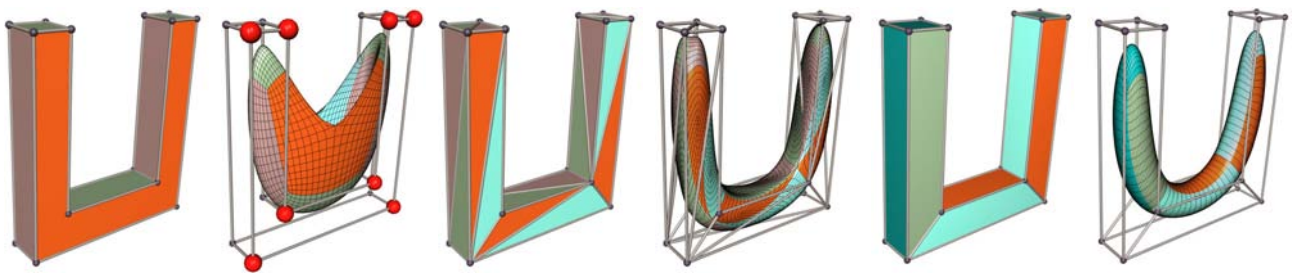


Figure 3.24: Non-convex base face. Subdivision sees the front-facing polygon as an 8-gon (red balls) and treats it like any other such – disregarding the fact that the centroid lies outside the face polygon. To split the face into convex parts is the solution then, as triangles or quads, for instance.

Ripples with high valences. The great thing about subdivision is that it works with faces of any degree m and with vertices of any valence n . But unfortunately, with higher n or m , *ripples* appear on the surface. Fig. 3.25 shows an example. The first thing to note is that the phenomenon is basically the same for vertices and for faces: The face point of a degree n face becomes a valence n vertex after the first subdivision. So the problem is merely one of high-valence vertices. One apparent reason for the unwanted behaviour is shown in 3.25 (2a): The edge point from the first subdivision is the average of the two endpoints of the edge and the centroids of the two adjacent faces. The ripples clearly correlate with the deviation of these four points from a common plane: When they are nearly coplanar, the ripples mostly disappear – at the expense of reducing the curvature in one direction, of course.

The true reason however seems to be that the computation of vertex points and face points is not completely symmetric, as in Fig. 3.25 (2e,f): The curvature of the bent quad mesh is zero in one direction, which is not the case if the quads are split into two triangles: Their subdivision surface exhibits ripples.

3.2.4 Non-convex Faces

Subdivision works irrespective of the geometry of a face, especially also on faces that are not convex. Such faces may have the property that the centroid does not lie inside the face polygon, as is the case for the u-shape in Fig. 3.24. The solution to this problem is of course to split the nonconvex face into smaller faces.

The reason why non-convex faces are not automatically split into convex parts is of course the fact that for subdivision, “*connectivity matters*”. The decisions which vertices to connect plays a great role for the resulting subdivision surface. It is important for the symmetry of the model: The apparent U-symmetry is lost with the triangulation in Fig. 3.24 (c), but kept with the control mesh (e). Abstract properties like symmetry are in the responsibility of the artist; and the control mesh connectivity is a degree of freedom for expressive design.

3.2.5 Example Models

In the following research the promise that any closed manifold mesh can do as control mesh of a subdivision surface was taken literally! The example models are very clean models from the Viewpoint Corporation [Vie]. They have a vast collection of highly polished, commercially available models, and they offer a small collection of models as a trial package

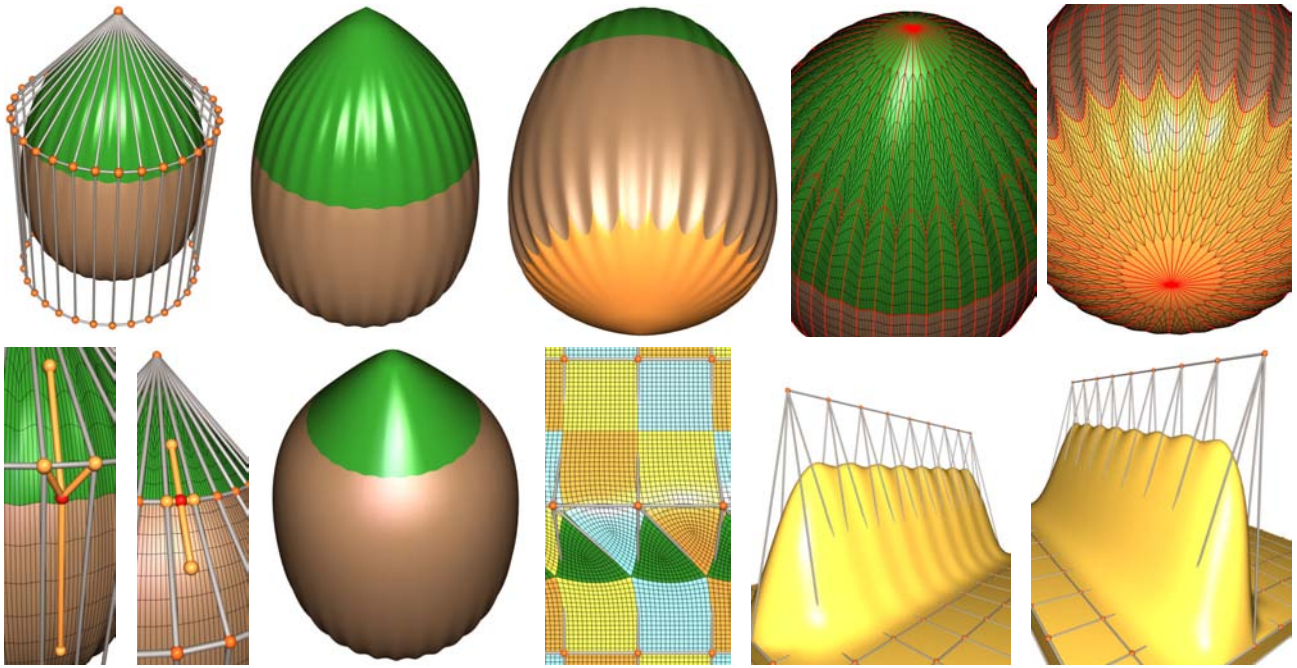


Figure 3.25: Example of *ripples* appearing with high valences. Top row, (1a,1c): A cylinder with a valence 32 vertex on top and a degree 32 face on the bottom. (1d,1e): The borders between the quads from the first subdivision are highlighted in red. The quads around the centers are much longer and thinner than at the periphery. Bottom row, (2a,2b): The edge point from the first subdivision is shown in red. (2c): The effect is reduced with the control mesh as in (2b). (2d): Subdivision surface from a regular quad grid where one row of quads is split into triangles. (2e-2f): The triangles have a ripple problem.

that can be obtained by registered users for free. These models were indeed of such a high quality that they could be used as control meshes right away. This is quite a valuable property.

Another interesting thing about these models is are not just made of triangles. Many other beautiful standard models used extensively in computer graphics like the Stanford bunny, the dragon or buddha models, are obtained from 3D acquisition methods. The Viewpoint models are instead clean synthetic models exported from CAD systems. They do not contain coplanar unconnected faces and the like. On the other hand, their use as subdivision surface control meshes is an abuse, because they were not conceived for this purpose – it is important to keep this fact in mind when some problems of their subdivision surfaces are discussed in the following.

The Viewpoint Girl model. The girl is made of six different shells, each topologically equivalent to the sphere: The legs, the arms, the body, and the head. Several shells in one control mesh, and also interpenetrations between different shells, are no problem for subdivision. It takes only the local 2-neighbourhood into account.

The girl is a great example that demonstrates the applicability of subdivision surfaces as a sculpting tool for smooth, natural forms – with particularly impressive results for the hands and the face. The hand is basically made from quad extrusions for the hand itself, and for the fingers. The quads are obviously diagonally split when they are not sufficiently planar. This mixed triangle-quad approach gives quite nice results, especially when the control mesh is so lean. The only visible artifacts are in the regions of the finger tips/finger nails: There is one vertex with very high valence (7 or 8) at every finger, which obviously attracts the surface and therefore leads to higher curvature.

The face is also a nice homogeneous mixture of triangles and quad faces, and it seems to avoid high valence quad faces. There are different artifacts here: It contains faces with different materials, and the borders between them are affected by surface shift, especially in the region around the mouth. The material borders are nicely modeled in the polygonal control mesh. But the border has triangles on one side, and quads on the other – and this leads to a border curve that has the same shape as already shown in Fig. 3.25 (2d).

The Viewpoint Cessna model. The cessna airplane model is basically a tubular shape made of quad extrusion with a very particular property: Some of the quads are grouped together to form a region, like the white windows made of three



Figure 3.26: The Viewpoint models **girl** (row 1), **cessna** (row 2), **dodge** (row 3), **airboat** (rows 4, 5), and **trumpet** (row 6), and their respective subdivision surfaces.

quads. These regions have a very thin border around it, made of thin, long quads, with one triangle in each corner. The green border around the windows is made in such way.

Such a configuration is subject to serious surface shift: The size of the border increases to the expense of the much bigger neighbouring faces. The amount of growth is proportional to the difference in size between border and neighbours. This is the reason why the right and left green borders are thicker than the border on top and bottom of the window: The white window quads are three or four times as wide as they are high.

These thin borders are also causing another, at first surprising, artifact: They decouple the interior of the region from the exterior of the surface. An example is the region on the nose of the airplane (Fig. [labelfig:viewpoint-models \(2b\)](#)), where the otherwise quite regular tessellation becomes more closely spaced on the region border. The 1-neighbourhood of the interior is the border, so the exterior of the region has no influence on the subdivision surface of the interior.

This must then lead to shape artifacts that are also noticeable when there is no change of material: There is a thin horizontal border where the door is, but the main direction of high curvature of the tubular shape is vertical – as a consequence, the surface exhibits a noticeably higher vertical curvature where the thin horizontal border is (Fig. [labelfig:viewpoint-models \(2e\)](#)).

The Viewpoint Dodge model. The dodge car model demonstrates how much subdivision surfaces depend on the regularity of their control meshes. When looking at the polygonal control mesh, the vertical front of the car looks perfectly regular and clean. Only closer inspection reveals that just a few edges are actually missing here and there, and therefore this portion of the control mesh is not perfectly symmetric.

The subdivision surface however amplifies such irregularities very much. They become especially noticeable by following the reflection lines and highlights on the surface, which are of course much more detailed when the model is subdivided and the tessellation has a high resolution.

The importance of maintaining symmetry is especially apparent with the bottom side of the car (Figs. [3.26 \(3e\)](#), [\(3f\)](#)): The control mesh is not symmetric, because the left half of the mesh contains one more edge than the – actually mirror-symmetric – right part. This one edge, because it is very long, leads to a quite serious asymmetry when subdivision is applied. As already mentioned, for subdivision “*connectivity matters*”.

The Viewpoint Airboat model. The airboat model is very interesting, as a special modeling technique has been used with it. Probably in order to improve the visual appearance from the edges of the model, many of them have been provided with parallel edges on both sides in a small, constant offset distance. On both sides, the offset edges lie in the same plane as the original face and its border. In the corners, this leads to a configuration called a “suitcase corner”.

As is clearly visible in Fig. [3.26](#) in the first two images ([4a](#)), ([4b](#)), this control mesh construction leads to really nice round beveled edges with apparently constant curvature. The reason is that the subdivision surface must follow the prescribed curvature in cross-edge direction, which is defined by the equally spaced sequence of offset edge, face edge, and offset edge. So this technique yields a similar effect as the semi-sharp creases from the Geri’s game paper by deRose et al. [[TDT98](#)], yet not by a modification of the subdivision rules, but alone by modifying the mesh connectivity.

Yet only three edges are not sufficient to decouple the face interiors. The price to pay is therefore a severe surface shift in the face interior, as in Fig. [3.26 \(4c\)](#). To reduce this effect, four edges would have to be used around the face borders, for instance a bevel (two edges) plus one offset edge on each side. This would decouple the interiors of the faces on both sides. But when the distortion of the tessellation is hidden, for example through assigning the same material everywhere, very good results are possible with this technique - see for instance with the engine of the airboat in Fig. [3.26 \(4d-e\)](#).

This approach leads to serious problems, however, when the corner angle is more than 180 degrees, such as the 270 degrees in Fig. [3.26 \(5a,b,c\)](#). The very small (green and cyan) triangles in the corner, where the offset edges cross, are extremely stretched. The reason is that the valence 6 vertex, at the point where the offset edges actually cross, is very much dragged – but its position is plausible to be the centroid of its neighbours, which are relatively far away. This and the next two images, ([5d](#)) and ([5e](#)), make it very clear that an artist, when modeling with subdivision surfaces, should take great care about relative sizes of neighbouring control mesh faces.

The Viewpoint Trumpet model. The trumpet model represents a musical instrument that is one of the greatest fields where subdivision can be applied: A trumpet has, by its nature, a tubular shape. In this case it is even a completely regular quad extrusion, which is the parade discipline of Catmull/Clark surfaces, the regular B-spline control grid. This explains for example why the surface from the mouth piece is so beautiful and high-quality (Figs. [3.26 \(6b,6c\)](#)).

The limitations of a purely tube-based approach are the connection of crossing tubes. This problem was ignored here, and the tubes are allowed to penetrate each other. Another problem that was not visible at first becomes apparent with subdivision: The valves are actually composed of three unconnected shells. Subdivision of course requires neighbourhood information, and a properly connected base mesh, to create a connected surface – which adds to the list of mesh integrity properties required for subdivision.

3.3 Options for Adaptive Tessellation and Display of Subdivision Surfaces

The great problem with the recursive refinement of subdivision surfaces is that the number of surface primitives grows exponentially with the levels of refinement. The constant factor is 4 for both the Catmull/Clark and Loop schemes: One base face first leads to 4, then to 16, to 64, and finally to 256 faces on the fourth refinement level. This makes for 1536 little quads from an object as simple as a cube! Computer memory becomes more affordable every day, so the size of these data is less of an issue – but is it actually reasonable to display such high levels of refinement? A simple consideration gives an answer:

- A typical computer display has a resolution between 1024×768 and 1600×1200 , so roughly 1-2 million pixels.
- About half of the faces of a simple 3D object are visible if there is not too much self-occlusion.
- Each (quad) face corresponds to 256 quads on the fourth refinement level.
- Consequently, a four times refined object with just $2 \cdot 2,000,000/256 = 15625$ base mesh quads will on average (!) display more quad faces than there are pixels on the screen.

For the purposes of interactive display, it is clearly an overkill to display one quad for each pixel – except at the object silhouette, or in regions with great color changes (e.g., highlights), simple Gouraud shading can do the job much cheaper without loss of quality. For other applications though, such as producing a physical realization of a virtual 3D shape (“*physical mockup*”, “*rapid prototyping*”), a very accurate tessellation is vital. In these cases a uniform tessellation may be inevitable.

3.3.1 Recursive Subdivision with a Hierarchical Data Structure

The above argument lead our group to search for *output sensitive* tessellation methods. The objective of computer graphics is image synthesis. Within this domain, the parade field of output sensitivity is *raytracing*: Not are the objects rasterized, instead each pixel actively looks for the relevant objects in the scene that contribute to its color value. This avoids the problem of having to know *a priori* the pixel coverage of each object displayed, and the annoying fact that also invisible objects need to be rasterized, if not lit, as is the case when using graphics hardware. Heavy overdraw is a problem for the rasterization hardware, but not for raytracing. With a hierarchical spatial data structure the closest object along the ray can in principle always be found in logarithmic time, with respect to the number of objects in the scene. Hardware-supported rendering in principle proceeds by simply rendering all objects, so it has a linear complexity unless sophisticated culling is used. – Algorithms to compute the intersection of a ray with a subdivision surface all proceed basically like this:

- Assuming a ray from the eye intersects the bounding box of a subdivision surface,
- determine the base faces whose patches are most likely hit, and
- recursively subdivide them,
- until the (projected) bounding volume has about the size of the pixel, which counts as an intersection between ray and subdivision surface.

What is needed for any such intersection algorithm to work is a method for local refinement, as opposed to the aforementioned uniform, global, refinement. Consider again how the refinement proceeds:

- to compute a face point requires all vertices of the face
- to compute an edge point requires (a) the two vertices of the edge, and (b) the the two adjacent face points
- to compute a vertex point requires (a) all face points of the incident faces, and (b) all neighbour vertices.

Data structures for refinement to arbitrary depth. After the first subdivision all faces are quads, which is convenient for designing a data structure. So given the first subdivision, how should the subdivision surfaces be organized to support local refinement? The canonical way to represent a face is by an object with four references to the corner vertices, and another four references to child nodes. They represent the (potential) sub-quads, so the children are of the same data type as the face, yet the references are initially set to *nil*.

A critical issue is where to store the connectivity of the subdivision. Of course one might provide each face on any level with four references to its neighbour faces. But the space overhead is considerable, as the four children of a face are always arranged in the same way. So instead of providing each face with explicit neighbour pointers it is better to make use of the hierarchic structure of subdivision. A face simply asks its parent face where to find the respective neighbour. Two of the four neighbours are always siblings, so there is a 50% chance that the neighbour request can be answered by the direct parent. To find also the other two, the neighbour request is propagated up the hierarchy, and the result of the request back down. The upwards-propagation might eventually lead to a face from the first subdivision. But the parent of such a face is a base face. The base face must provide explicit neighbourhood information, otherwise it would not have been possible to compute the first subdivision. This approach trades space for time: Four neighbour references are saved, but in some cases it might be necessary to traverse all refinement levels up and down to answer a neighbour request.

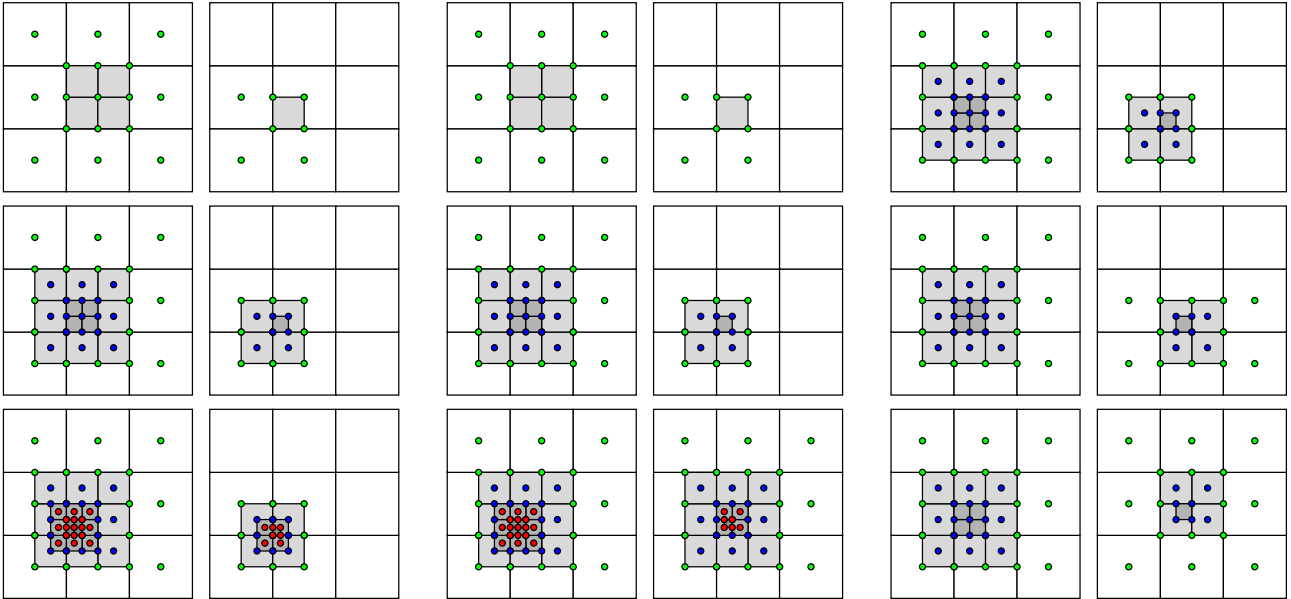


Figure 3.27: Complete vs. incomplete recursive refinement. Columns (a,c,e) show complete and (b,d,f), for comparison, incomplete refinement. Columns (a,b) show in rows 1,2,3 the refinement requests LL,LL,LL, which stands for 'lower left'. Columns (c,d) show the refinements LL,LL,UR in rows 1,2,3. And columns (d,e) finally compare the three basic cases LL,LL in (1e-f), LL,LR, which is symmetric to LL,UL, in (2e-f), and LL,UR in (3e-f).

A neighbour request can also lead to recursive subdivision when the requested neighbour, or even some of its parents, are not computed yet. Subdivision is triggered according to a simple rule that can be formulated in either of two ways, upwards or downwards the hierarchy:

- A face can only be subdivided if its full 1-neighbourhood exists.
- For each face the full 1-neighbourhood of its parent face exists.

As an example consider the initial state where only the first subdivision exists and all child references are *nil*. Each face f^1 from the first subdivision can be refined, as all faces incident to any of its four vertices $v_1^1, v_2^1, v_3^1, v_4^1$ exist. Let $f_1^2, f_2^2, f_3^2, f_4^2$ be the four children of f^1 on level 2, numbered in the same order as the vertices. Then to subdivide its child f_2^1 , for instance, further it is necessary to compute all faces incident to v_2^1 . So far only one of them exists, f_1 . – So this is an example of how a refinement request triggers subdivision on a higher, i.e., a parent level.

Complete and incomplete recursive refinement. There are basically two ways to organize the selective refinement process: In response to a refinement query either all children can be generated or only the requested one. The latter strategy, *incomplete refinement*, has higher administrative costs, but it can considerably save computations. Both strategies are compared in columns (a,b) in Fig. 3.27. A sub-face consists of one face point, two edge points, and one vertex point. To compute the vertex point, all face points must exist of the faces around the same vertex on one level higher, as in Fig. 3.27 (1b). This diagram shows the basic pattern for incomplete refinement, where three face points are requested in addition to the four points of the sub-face (in the regular case). Incomplete refinement performs better, e.g., with (3a) vs. (3b), but a child request can also lead to cascading computations on higher levels: In (3d) not much is saved compared to (3c). The reason is shown in (3f): The (LL,UR) request leads to computing the face points of all faces incident to the center face! The direct comparison reveals that that complete refinement scheme in columns (a),(c),(e) is more stable. Back-propagation always ends at one level higher, so it trades traversal costs against point computation costs. Which scheme to choose may be machine specific, because whether traversal is faster or the computations may depend on the actual hardware architecture.

A hierarchical data structure is not so good for interactive rendering. The author's first attempts to realize non-uniform recursive refinement, and actually the first interactive demos on subdivision surfaces in our group, were based on the 'incomplete refinement' strategy. The resulting data structure was called the SMesh, S for subdivision, and demos based on it have been shown with some success for a while. But it turned out that this is not the optimal approach for interactive display, mainly because of three reasons:

- An adaptive tessellation without cracks requires nasty additional administrative overhead,
- Each single quad is rendered as an individual, which is inefficient
- When the base mesh is changed it is not easy to find and update all affected entities.

To avoid cracks all faces need to be projected to the limit surface. But the limit position of a vertex v depends on all vertices from all faces incident to v , which is an awfully complete neighbourhood! Furthermore the completion of the neighbourhood triggers new vertex requests, and these can lead again to (cascading) subdivision. Consider for example an edge point of the LL center quad in Fig. 3.27 (1b). Only 5 of the 3×3 points needed for computing the limit positions are present, three edge and one vertex point are missing; the latter in turn requires two new face points, and so on – and quickly the attractive sparseness is abandoned. Not to mention the fact that for each vertex, both the subdivided position and its limit projection need to be stored, as further refinement is always possible – which doubles the data size.

A hierarchical data structure is very good for accurate rendering. While these issues have prevented the use of SMeshes, e.g., for interactive 3D modeling, they are not grave when using them for a different purpose: Raytracing, where the refinement resolution is driven by the pixel error. The SMeshes have inspired Thorsten Techmann from our group, who extended the basic principle considerably in his diploma thesis to cover both Catmull/Clark and Loop surfaces with basically the same refinement mechanism. Kerstin Müller and he later took adaptive refinement to great success with their *ShaOLin* (“Shadow Of the Line”) algorithm [MTF03]: Among other techniques they also introduced a *refinement cache*. It keeps all the tiny refined faces for a while, i.e., during the computation of a few rows of the image. This is very effective with raytracing since pixel coherence leads to many similar evaluations in more or less the same surface regions.

3.3.2 Direct Evaluation: Parametric Surfaces and Basis Functions

The recursive approach can hardly keep up when the control mesh is interactively changed, due to its considerable administrative overhead: In each frame, to just re-compute a two times subdivided face, $4 + 4 \cdot 4 = 20$ little faces must be deleted, allocated, $5 \cdot 5 = 25$ vertices are allocated, computed, projected, and references to them are put into the 20 faces by recursive traversal. Why not simply compute just the 25 limit vertices? And for adaptive tessellations it would even be more profitable if any vertex that is needed could be evaluated directly – without depending on any caches or other complicated data structures. There are basically two ways to directly obtain a specific vertex: Parametric evaluation, and the extraction of weights from basis functions.

Parametric subdivision surfaces. Direct parametric evaluation of subdivision surfaces was presented by Jos Stam for triangle- and quad-based subdivision, in both cases following the same principle [Sta99, Sta98]. It starts not at the first but at the second subdivision, which also induces a partition of the subdivision surface into quadrangular free-form patches. Each of these patches has a canonical $[0, 1] \times [0, 1]$ parametrization, and furthermore every patch has at most one irregular vertex (cf. Figs. 3.10 and 3.17). W.l.o.g. it is at parameter value $(0, 0)$. As a consequence, 3 from 4 of its sub-patches (now on level 3) have a regular 4×4 local control mesh; and *any* point on the surface with parametric coordinates $(x, y) \in ([0, 1] \times [0, 1] \setminus [0, \frac{1}{2}] \times [0, \frac{1}{2}])$ can be directly obtained by just evaluating the respective bi-cubic tensor product B-spline surface! In order to obtain points on the remaining quarter $[0, \frac{1}{2}]^2$ of the patch, it is recursively split, and the same approach is applied again – then leaving only $[0, \frac{1}{4}]^2$ as problematic region on the 4th level. For any parameter value $(x, y) \in [0, 1]^2$ except $(0, 0)$, this recursion will come to an end. The irregular point in $(0, 0)$ is of course obtained directly, using the limit position rules from sec. 3.1.5.

This is a very elegant approach, and the canonical method to practically obtain points on the subdivision surface whose parametric coordinates are not inverse powers of two, i.e., that are not of the form $(2^{-m}, 2^{-n})$, $m, n \geq 0$. This is important for algorithms that do not deal with a given (regular) sampling of the surface, but need to actively sample the surface at general parameter values. To summarize, this method is very flexible, but not especially efficient: To evaluate a point on a B-Spline surface requires five curve evaluations (see sec. 3.1.3 and Fig. 3.13).

Evaluation of basis functions. Most surprisingly, any kind of recursive evaluation can actually be skipped! – But only when the parametric coordinates of the points that will be requested are known beforehand. To perform one subdivision step means to compute new points as convex combinations of existing points. The new points are in turn the CVs of the next level. So it is easily possible to track and accumulate the weights, for each point on any given level k , all the way from the base CVs to any level k . This idea applies also to the limit rules, so that each vertex of the tessellation can be directly obtained from the base vertices. These weights are therefore called the *direct weights* of a tessellation vertex.

This is shown for the regular setting in Fig. 3.28. The direct weights can also be directly obtained, by regarding the patch as a mapping $\mathbb{R}^2 \rightarrow \mathbb{R}$, more specifically $[0, 1] \times [0, 1] \rightarrow [0, 1]$. Given a point on the limit surface with parametric coordinates (x, y) the influence of a particular CV on this point can be measured simply as the height of the point above

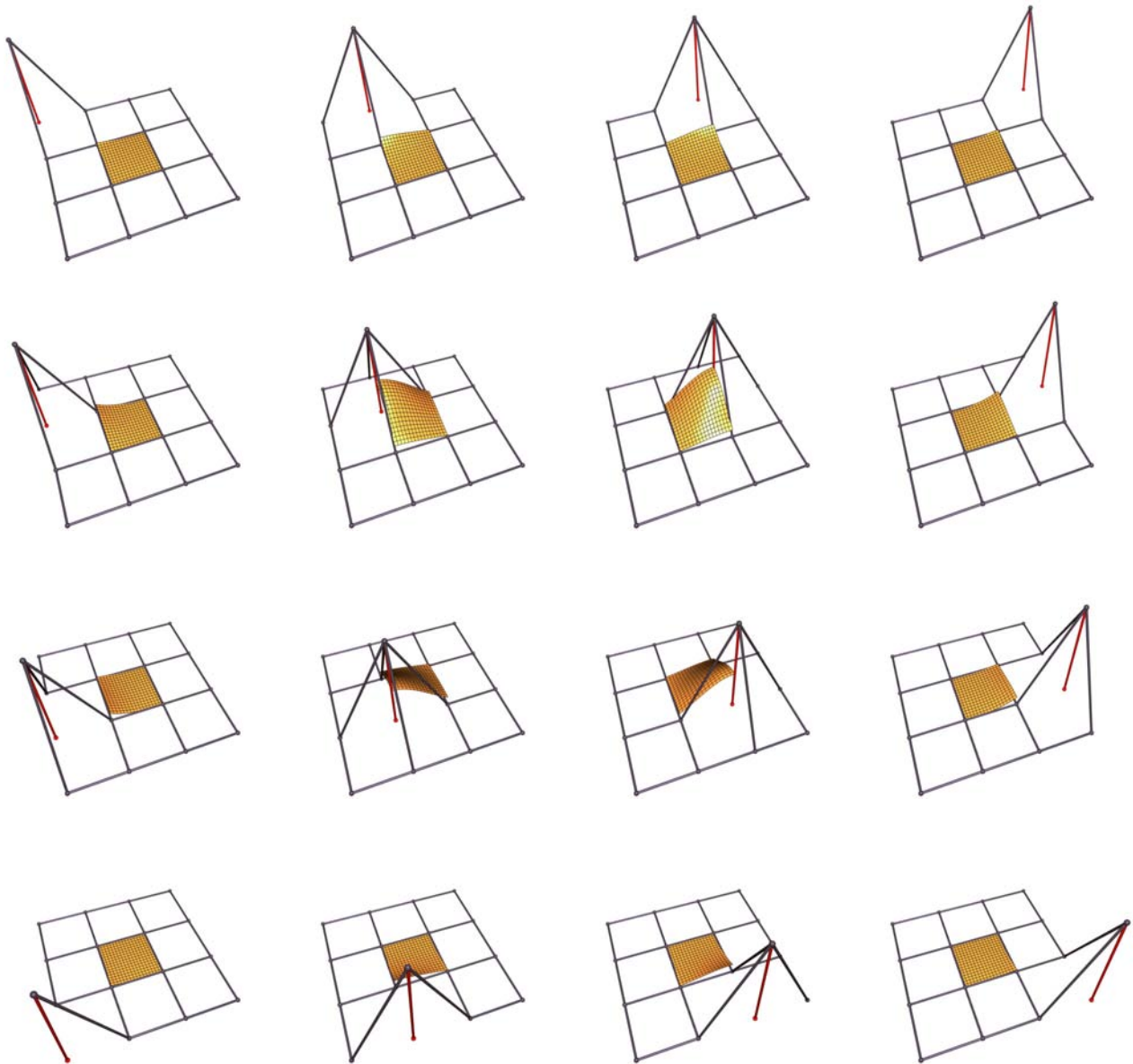


Figure 3.28: Basis functions of a regular patch. Each basis function corresponds to setting the z -value of one CV to 1 while all other CVs are at height 0. For a specific point on the limit surface with parameter values $(x, y) \in [0, 1] \times [0, 1]$ the vertical displacement then measures the influence the respective CV has on this point. The sixteen *direct weights* of a limit point with respect to all CVs sum to one, at least in theory. Points from the k -th subdivision level have parametric coordinates of the form $(i/2^k, j/2^k)$, with $i, j = 0, \dots, 2^k$. For these points, the direct weights equal the accumulated weights of the subdivision rules across all levels.

$$\text{face} := (v0, v1, v2, v3) \rightarrow \frac{1}{4}v0 + \frac{1}{4}v1 + \frac{1}{4}v2 + \frac{1}{4}v3$$

$$\text{edge} := (v1, v2, f1, f2) \rightarrow \frac{1}{4}v1 + \frac{1}{4}v2 + \frac{1}{4}f1 + \frac{1}{4}f2$$

$$\text{vertex} := (n, v, f, e) \rightarrow \frac{(n-2)v}{n} + \frac{(\sum_{j=1}^n e_j) + (\sum_{j=1}^n f_j)}{n^2}$$

```

CC_grid_refine := proc(N, A, k)
local M, B, i, j, ii, jj;
M := 2 * N - 3;
B := array(1..M, 1..M);
for i to N - 1 do
for j to N - 1 do B[2*i-1, 2*j-1] := face(A[i, j], A[i, j+1], A[i+1, j], A[i+1, j+1]) od;
od;
for i to N - 2 do for j to N - 2 do B[2*i, 2*j] := vertex(4, A[i+1, j+1],
[A[i, j+1], A[i+2, j+1], A[i+1, j], A[i+1, j+2]],
[B[2*i-1, 2*j-1], B[2*i+1, 2*j-1], B[2*i-1, 2*j+1], B[2*i+1, 2*j+1]])
od;
od;
for i to N - 1 do for j to N - 2 do
B[2*i-1, 2*j] := edge(A[i, j+1], A[i+1, j+1], B[2*i-1, 2*j-1], B[2*i-1, 2*j+1]);
B[2*j, 2*i-1] := edge(A[j+1, i], A[j+1, i+1], B[2*j-1, 2*i-1], B[2*j+1, 2*i-1])
od;
od;
RETURN(B)
end

CC_grid := proc(k::integer)
local B, n, v, vv, i, j, l, BK;
n := 4;
B := array(1..4, 1..4,
[[v11, v12, v13, v14], [v21, v22, v23, v24], [v31, v32, v33, v34], [v41, v42, v43, v44]]);
for i to k do B := CC_grid_refine(n, B, k); n := 2 * n - 3 od;
RETURN([n, evalm(B)])
end

p4,4 := CC_grid(3)[2][4, 4];

p4,4 :=


|                          |                           |                           |                           |                          |                           |                        |                         |
|--------------------------|---------------------------|---------------------------|---------------------------|--------------------------|---------------------------|------------------------|-------------------------|
| $\frac{1225}{262144}v11$ | $+$                       | $\frac{11025}{262144}v12$ | $+$                       | $\frac{5635}{262144}v13$ | $+$                       | $\frac{35}{262144}v14$ |                         |
| $+$                      | $\frac{11025}{262144}v21$ | $+$                       | $\frac{99225}{262144}v22$ | $+$                      | $\frac{50715}{262144}v23$ | $+$                    | $\frac{315}{262144}v24$ |
| $+$                      | $\frac{5635}{262144}v31$  | $+$                       | $\frac{50715}{262144}v32$ | $+$                      | $\frac{25921}{262144}v33$ | $+$                    | $\frac{161}{262144}v34$ |
| $+$                      | $\frac{35}{262144}v41$    | $+$                       | $\frac{315}{262144}v42$   | $+$                      | $\frac{161}{262144}v43$   | $+$                    | $\frac{1}{262144}v44$   |


```

Figure 3.29: Maple code to derive direct weights symbolically. The function call $CC_grid(3)[2]$ yields an 11×11 matrix that is produced from a 4×4 base mesh after three refinement steps. The example shows the weights for direct subdivision of point $p_{4,4}$ from the 3rd refinement level, which corresponds to parameter values $(\frac{1}{4}, \frac{1}{4})$ on the patch. This point still needs to be projected to the limit surface to eventually yield the direct position weights. – Note that this procedure works only for the *regular* case, but with a slight modification it can be generalized to arbitrary valences: Simply assume that on every subdivision level, the points in the top left and bottom right corners have been computed elsewhere. On the 3rd refinement level, the points in these potentially irregular corners are $p_{1,1}, p_{1,2}, p_{2,1}, p_{2,2}$ and $p_{10,10}, p_{10,11}, p_{11,10}, p_{11,11}$. If they are treated as symbolic constants, the symbolic algebra approach works as before and produces valence-independent direct weights. This idea is further elaborated in the section 3.4 on *vertex and face rings*.

the ground: Set only the height of this CV to one, and all other CVs to zero. – When the direct weights with respect to all CVs are known, the limit point at (x, y) of any patch can be directly obtained by a weighted sum over the patch CVs. To practically determine the direct weights a software package for symbolic algebra, Maple [Map], was used. With a few lines of code Maple computes the weights for any desired refinement level (see Fig. 3.29).

This is an extremely attractive way of evaluating a subdivision surface, because it perfectly realizes the idea of *tessellation on demand* for adaptive real-time display: Given a coarsely approximated patch that needs to be displayed at higher resolution, e.g., because the observer comes closer, just compute some more points refine the tessellation. No administrative overhead is involved, and also changes in the control mesh can easily be handled: Just recompute a few points on the affected patches for a coarse resolution and successively refine in the following frames. This spreads the tessellation cost over time, and maintains instant feedback during the actual modeling. So this approach has been pursued next, and for interactive applications it worked much better than the SMeshes.

Drawbacks of the evaluation with direct weights. Our experiences identified three remaining issues with this method:

- **Fixed maximum resolution and maximum vertex valence.** When starting from the first subdivision, two from four points of each quad may be irregular. As the subdivision rules are valence-dependent, there is basically one set of direct weights for each valence. This makes for an $m \times m$ table of valence combinations, each with its own set of direct weights for each tessellation vertex. Most annoying, the maximum valence m is pre-determined when setting up the table. Yet this problem is solvable (see below).

The maximum resolution of the tessellation is pre-determined as well, by the choice of a set of parametric coordinates for which direct weights are derived. This is typically a maximum subdivision level k with points $(i/2^k, j/2^k)$ where i and j vary from 0 to 2^k .

- **High operation counts.** Most points of the tessellation depend on *all* base vertices in the 1-neighbourhood of the face! For the regular case, this means that each point in the tessellation is a convex combination of 16 base vertices. To approximate a patch by 8×8 quads, 9×9 vertices must be computed. This makes for $81 \cdot 3 \cdot 16 = 3888$ float multiplications for the weights and $81 \cdot 3 \cdot 15 = 3645$ additions – for one patch! This can be reduced a bit if zero weights are not multiplied, which occur only at the patch border, though.

Surprisingly, to obtain the surface normals, in fact *twice* as much work must be invested as for the positions! The reason is that the surface normal is the cross product of the limit tangent vectors – and each tangent vector is also a weighted sum of the base CVs. The only difference is that the tangent weights sum to zero instead one. In total, this makes for more than **11.000 floating point multiplications** and about as many additions to produce the 81 points and 81 normals!

- **Precision problems.** Unfortunately the accumulated weights do not sum exactly to one any more. A second effect is that when summing up sixteen float numbers, up to four bits are lost in float precision – but not always the same bits! Which bits are lost depends on the order in which the sixteen values are added. This leads to pixel errors on the patch borders: The border vertices from two patches do not coincide, even though the identical CVs and weights are used for both, only because the computations were performed in different orders.

In an attempt to tackle the problem of high operation counts, another feature from Maple [Map] was used: It can generate optimized C-source code for a given symbolic expression. The optimizer tries to reduce the operation count by identifying common sub-expressions, so that intermediate results are stored in temporary variables if they are used more than once. For the cited case of a 9×9 tessellation with 81 points, Maple has identified 244 common subexpressions for the full computation of limit points and normals – which gave some saving, but no fundamental improvement. Due to the reported problems, we never chose to actually publish our experiences and results with direct weights.

One paper though pursued the direct weights approach with success: Bolz and Schröder [BS02b] managed to solve the efficiency problems by a clever use of the streaming extensions of Intel CPUs, today available from AMD as well. The SSE hardware permits to execute four floating point operations at the same time [Int99a]. When the memory layout of the input data is arranged a suitable way (Array of Structs/Struct of Array issue, also called AoS/SoA issue), it is possible to keep the SSE unit on the processor all the time busy. This gives a performance boost over our (in this respect) admittedly simplistic implementation that uses only the standard math library. Bolz and Schröder reported that 302 base quads could be tessellated to level four, resulting in $302 \cdot 256 = 77312$ quads, in only 14 ms on a Pentium 4 with 1700 MHz. They also solved the precision problems, simply by copying border vertices from a patch to its neighbour patch to make sure that both use exactly the same data.

There exists an elegant way to also circumvent the first problem, the valence dependency of the direct weights, so that just one set of direct weights is sufficient. This technique, the *vertex and face rings*, is presented in the next section, along with a computation scheme that is optimized very much with respect to operation counts.

3.4 Adaptive Tessellation on-the-fly of Catmull/Clark surfaces with Crease Edges

After the first subdivision of the base mesh all faces are regular quads. Irregularity can only come from the vertices: Each quad face has one point that is a vertex point and another point that is a face point, and both of them are possibly irregular. The other two points are edge points, which are always regular, i.e., they have valence four. The core idea for speeding up the patch tessellation is to divide the computation in two parts: The potential irregularity is captured by *vertex* and *face rings* that are refined separately first. These rings are subsequently fed into the computation of the actual tessellation, which is then completely regular, and can therefore be optimized a lot.

This section presents an optimized version of the approach from the journal paper with the same title [Hav02b].

3.4.1 Vertex and Face Rings

Let $v \in V$ be a base vertex that belongs to a smooth base face $f \in F$ with degree n from the unrefined control mesh (V, E, F) with vertices V , edges E , and faces F . In terms of notation, the entities of the mesh are considered just elements of these sets. There is a distinction between a vertex v , which is a node of a graph, and its 3D position, which is referred to as v^0 . This notation is convenient for subdivision, where the base mesh is referred to as 0-th subdivision level. So in the following, superscripts refer to the 3D points attached to the respective entities (see also the mesh definition 2.30).

The face point $f^1 = \text{facepoint}(f)$ is the centroid, i.e., the average of the vertices v_0^0, \dots, v_{n-1}^0 of f . The vertices are ordered in *counterclockwise* orientation, but the sequence has no canonical start point; so in fact any vertex of f can do as v_0 . The first subdivision introduces edge points e_0^1, \dots, e_{n-1}^1 on every edge e_i . An edge point $e_i^1 = \text{edgepoint}(e_i)$ is the average of four points: f^1 , the face point of the adjacent face, and the endpoints v_i^0, v_{i+1}^0 of the edge. Please recall that the vertex sequence is cyclic, so vertex indices are always modulo n and, accordingly, edge e_{n-1} goes from v_{n-1} to v_0 .

Definition 3.4 (Face Ring) A face ring is defined by the face point f^1 together with the sequence $(e_0^1, v_1^1, \dots, v_{n-1}^1, e_{n-1}^1, v_0^1)$ of alternating edge and vertex points from the first subdivision. Face rings, like vertex rings, always start with an edge point. Consequently, the vertex point v_0^1 is the last point of the face ring.

Given a face ring, the limit position and normal of the face point f^1 can readily be computed (section 3.1.5 with limit stencil in Fig. 3.11 (a)). Consequently, all points f^2, f^3, \dots on any higher subdivision level can also be computed, now using the vertex rule: The face ring defines the first level quads around f^1 . The same ring on the next level is the alternating sequence of the face centroids and edge points from the present ring. Together with applying the vertex rule to f^1 then the *refined face ring* on the next subdivision level is obtained. See Fig. 3.30 for an example of face rings, as well as vertex rings, which are explained next.

Unlike with face rings, where all edges incident to the face point f^1 are smooth, some edges of a vertex v^1 may actually be sharp. The polygon ring around the vertex is therefore ordered according to the vertex classification. In any case in the following definition the polygon is arranged so that it always starts with a sharp edge, in case there is one. If there are more than one, as for crease and corner vertices, then also more than one vertex ring may be attached to v .

Definition 3.5 (Vertex Ring) The vertex ring of a vertex v with valence m is made of the first level vertex point v^1 and the polygon $(e_0^1, f_0^1, \dots, e_{m-1}^1, f_{m-1}^1)$ of alternating edge and face points around v . The vertex ring is again organized as an array of 3D points, but this time in clockwise direction. A vertex with sharp edges may have more than one vertex ring, according to the vertex classification and the following rules.

- If v has 0 sharp edges it is a **smooth vertex**, and the vertex ring is closed (see Fig. 3.11). The vertex ring array starts with an arbitrary edge point, and it ends with a face point.
- If v has 1 sharp edges it is a **dart vertex**, and its vertex ring is closed as well. Its vertex ring array starts with the edge point of the single sharp edge incident to v .
- If v has 2 sharp edges it is a **crease vertex**, and it has two vertex rings. Both corresponding arrays contain an odd number of points. Each ring starts at one sharp edge, and the last point is the edge point of the other sharp edge.
- If v has ≥ 3 sharp edges it is a **corner**, and it may have an arbitrary number of vertex rings: There is one vertex ring for each sequence of consecutive smooth edges, and each ring starts, and ends, with a sharp edge.

A vertex or face ring represents a possibly irregular vertex with its 1-neighbourhood. Such a ring of quads contains enough information to be refined: The face points become the indirect neighbours on the next level, the edge points become the direct neighbours, and the next level vertex point can be computed as well. Note that a vertex has one surface normal with respect to each attached vertex ring. A vertex ring also equals a *wedge* from the corner vertex rule (see 3.1.6).

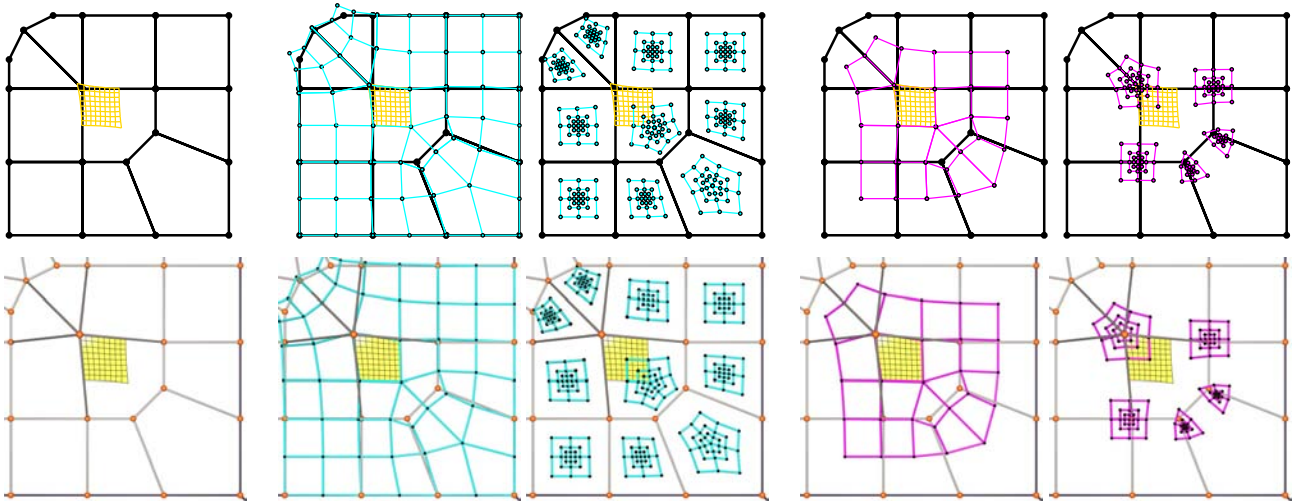


Figure 3.30: Vertex and Face Rings. Upper two rows: Schematic and perspective views of vertex and face rings. Left column: The control mesh with a pentagonal center face, partitioned into 5 quads by the first subdivision. Each quad corresponds to a patch, one of them is marked in yellow. Columns 2,3: Each face has a face ring made of the points from the first subdivision: The face point, and the sequence of edge- and vertex points. The face point valence equals the face degree. Column 3: Each ring is recursively subdivided, yielding the refined rings on levels 2,3, and 4. Column 4: Vertex rings around the vertices of the center face, with points from the first subdivision. Column 5: Each vertex ring is also refined, yielding the direct neighbourhoods of the (potentially) irregular vertices on levels 2,3, and 4.

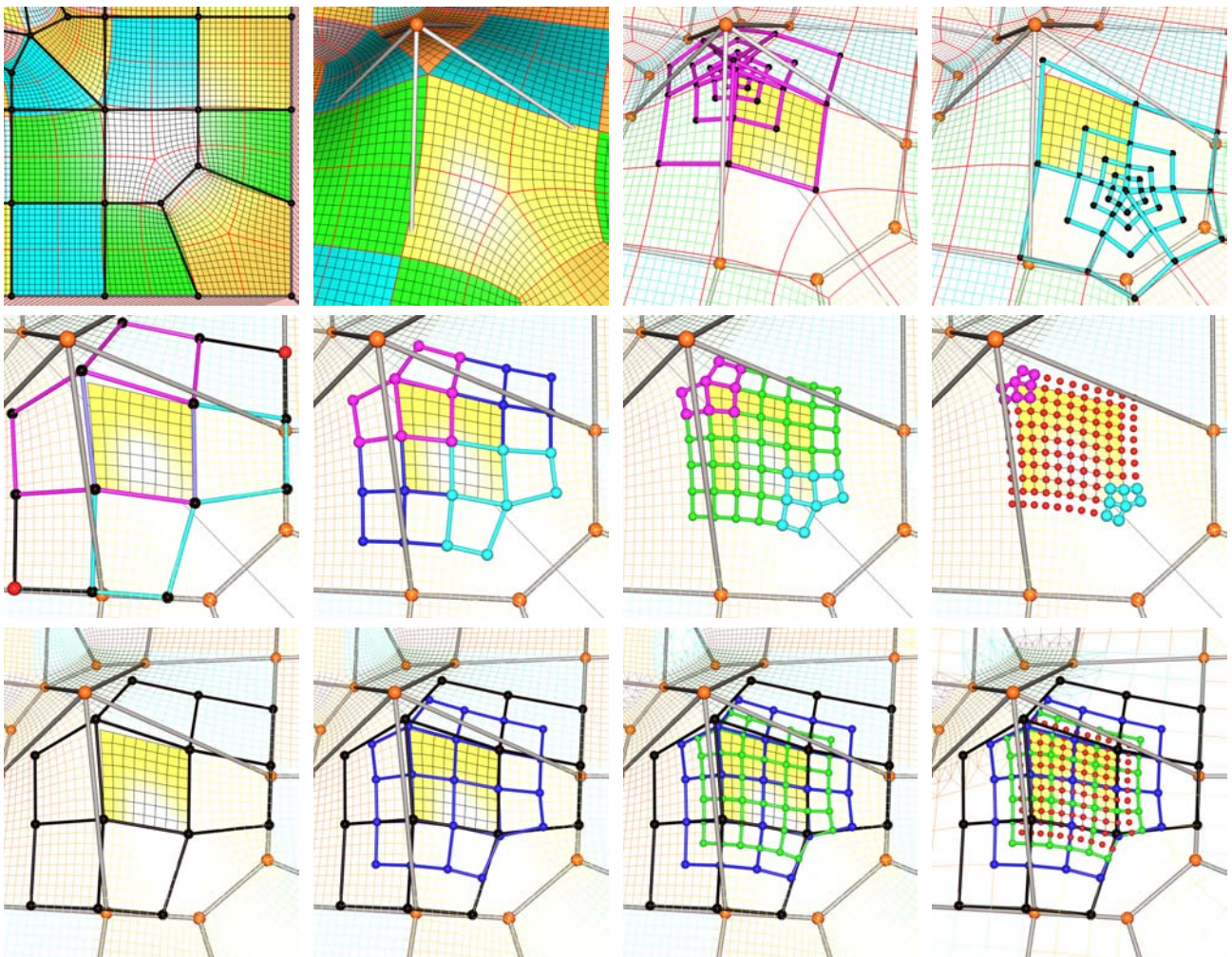


Figure 3.31: Recursive subdivision of a patch, and points fed in from vertex- and face rings, explained in 3.4.2.

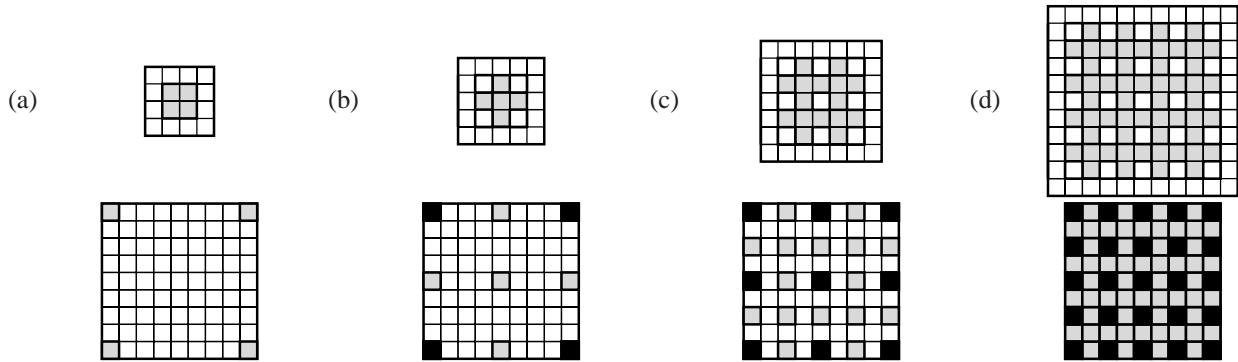


Figure 3.32: Pattern of projected points per level. Squares in the top row stand for points of the subdivision, the bottom row shows fixed 9×9 grid of tessellation data, initially containing garbage (white). (a) The first subdivision determines only the corners of the patch tessellation. (b) Only 5 points from the 2nd level need to be projected (grey) to obtain a 3×3 tessellation. (c), (d) develops the projection pattern: Only three points from each 2×2 block of refined points need to be projected, one already is.

3.4.2 Optimized Recursive Subdivision

The tessellation of each patch is stored in two arrays of 3D points with fixed size $(2^k + 1) \times (2^k + 1)$, one for vertex positions and one for surface normals. The size of these arrays is determined by an arbitrary but fixed maximum subdivision level $k = k_{\max}$. In the following considerations, $k_{\max} = 4$ is assumed. This facilitates the presentation of the technical details, but the approach applies analogously to other values of k_{\max} . It is equally possible, and even perfectly reasonable, to have several instances of the technique, each with different values of k_{\max} , in the same application. To set $k_{\max} = 4$ is a good choice, however, as further explained in sec. 3.4.5 below. This makes for arrays of size $9 \cdot 9 = 81$, totaling $2 \cdot 81 \cdot 3 \cdot 4 = 1944$ bytes per patch to store the positions and normals when three single precision 32 bit IEEE floating point numbers are consumed by each 3D point and normal vector.

Feeding in vertex and face rings. The vertex and face rings from the previous section help to make the computation of the actual tessellation highly regular, so that only the regular versions of subdivision rules, limit position rules, and limit normal rules must be considered. To see why this is so see Fig. 3.31, the key figure for understanding the process. The top row 1 of Fig. 3.31 shows the tessellation from subdivision level 4, projected to limit surface, in an overview (1a) and as close-up (1b). The patch in the upper left corner is incident to one face point and one vertex point, both of them are irregular with valence 5. The vertex and face rings of this patch are shown in (1c) and (1d). Note how the ring points correspond to the limit points from the tessellation.

Images (2a,3a) in Fig. 3.31 show the *local control mesh* of the patch on level 1. Without the irregular top left and bottom right this is a *quasi regular* 4×4 grid, only with two points missing. All points except the top right and bottom left points can be read from the vertex- and face rings. These two points, called p_{TR} and p_{BL} , are needed as additional data for a complete local patch control mesh. The next images (3b-3d) show how the local control mesh is refined, yielding an ever finer quasi-regular grid. One outer ring of vertices outside the patch border is always necessary for further refinement. As before, the top left and bottom right points are missing from the grid. But these points are not needed for grid refinement: On every level, the eight points in the top left and in the bottom right corners of the grid have already been computed from the vertex and face rings (2b-2d). Only the regular rest of the grid on every level is left to be computed.

Optimized subdivision and limit projection of the quasi-regular grid. Each level of refinement consists of two passes: First the grid is refined, then some points are projected on the limit surface. The refined grid is used for further subdivision on the next level, while the limit points successively fill the fixed-size tessellation.

The tessellation is initialized with four points from the first subdivision level: Two of them are the vertex and face points, whose limit positions and normals are available from the vertex and face rings. The other two are the (regular) edge points from the first subdivision, and they can be projected using the regular stencils. These four points and normals are put into the corners of the tessellation, in array positions 0, 8, 72, and 80, shown in grey in Fig. 3.32 (a). After this initialization a coarse level 1-approximation of the subdivision surface can already be displayed, with one quad per patch. The application may determine, however, that some patches deserve higher accuracy, due to some measures like projected screen size, surface curvature, etc. Then the tessellation is progressively filled on demand with additional limit surface points, as shown in Fig. 3.32. The two passes for refinement and limit projection are explained in detail next.

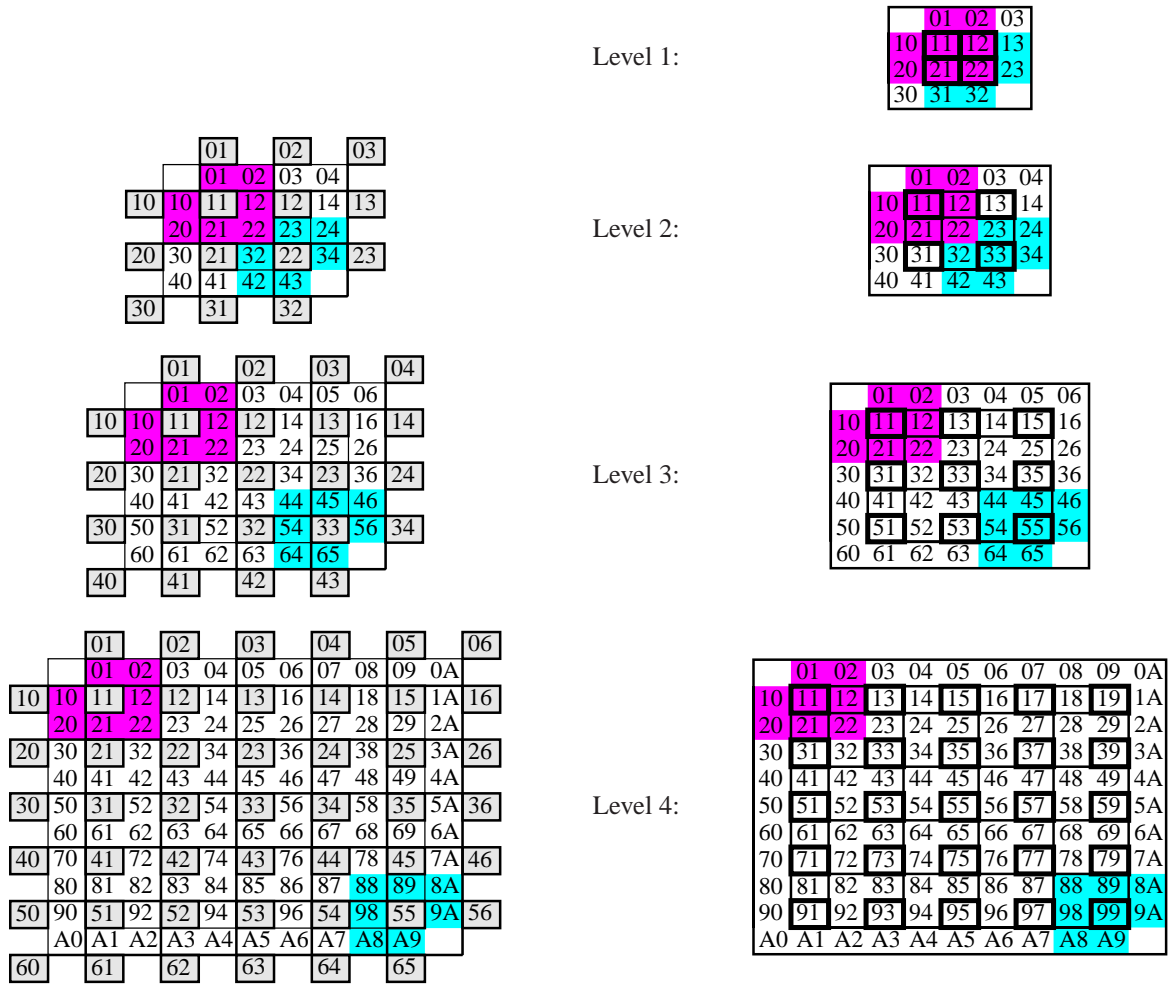


Figure 3.33: Subdivision and limit projection patterns. Left column: Pass 1, subdivision. Coarser levels are overlaid to finer levels, with corresponding (vertex) points on top of each other. The points fed in from vertex- and face rings are in the top left and bottom right (magenta and cyan). The 2×2 boxes show the pattern for applying the subdivision rule from Fig. 3.34. (2a): Level 1 \rightarrow 2. (3a): Level 2 \rightarrow 3. (4a): Level 3 \rightarrow 4. Right column: Pass 2, limit projection to obtain the pattern in Fig. 3.32. The grids (1b)-(4b) show the subdivision levels 1-4, exactly corresponding to Fig. 3.31 (2a)-(2d). The 2×2 boxes show the pattern for applying the limit projection rule from Fig. 3.35. – The letter ‘A’ is used as digit for number 10 to keep the diagram readable.

First pass: Grid subdivision using the SUBDIVISIONRULE. The right column from Fig. 3.33 shows the local quasi-regular control grid on levels 1-4, with the patch vertices surrounded by the outer ring of vertices needed for refinement. On level k the grid size is $(2^{k-1} + 3) \times (2^{k-1} + 3)$ and the tessellation has size $(2^{k-1} + 1) \times (2^{k-1} + 1)$, see Table 3.3.

Now consider the left column of Fig. 3.33. Diagram (2a) shows how level 2 is computed from level 1. The older level is doubly spaced and laid on top of the new level. The vertex and face rings blend in, and only eight points in the TR and BL corners remain to be computed. Diagram 3.33 (4a) goes from level 3 to 4, and it exhibits prototypically the regularity that can be exploited to optimize the subdivision. The double-spacing of the old grid induces a tiling into 2×2 blocks. These blocks are processed in row-wise order. The first row goes from block (13,14,23,24) to (19,1A,29,2A), the last row eventually finishes with block (95,96,A5,A6), digit ‘A’ meaning ‘10’. Half-blocks like (97,A7) are treated specially.

Level	Grid	Tessellation
1	4×4	2×2
2	5×5	3×3
3	7×7	5×5
4	11×11	9×9

Table 3.3: Grid sizes per level

Before the blocks can be processed in this order the grid must be initialized by computing the points 03-0A from the top. On the beginning of each new row of blocks the left border vertices (30-A0) are initialized first, together with two temporary variables, a' and b' . This is further explained in Fig. 3.34, where also the pseudocode of the SUBDIVISIONRULE function is listed. The block tiling seeks to make maximal use of common sub-expressions, and it introduces temporary

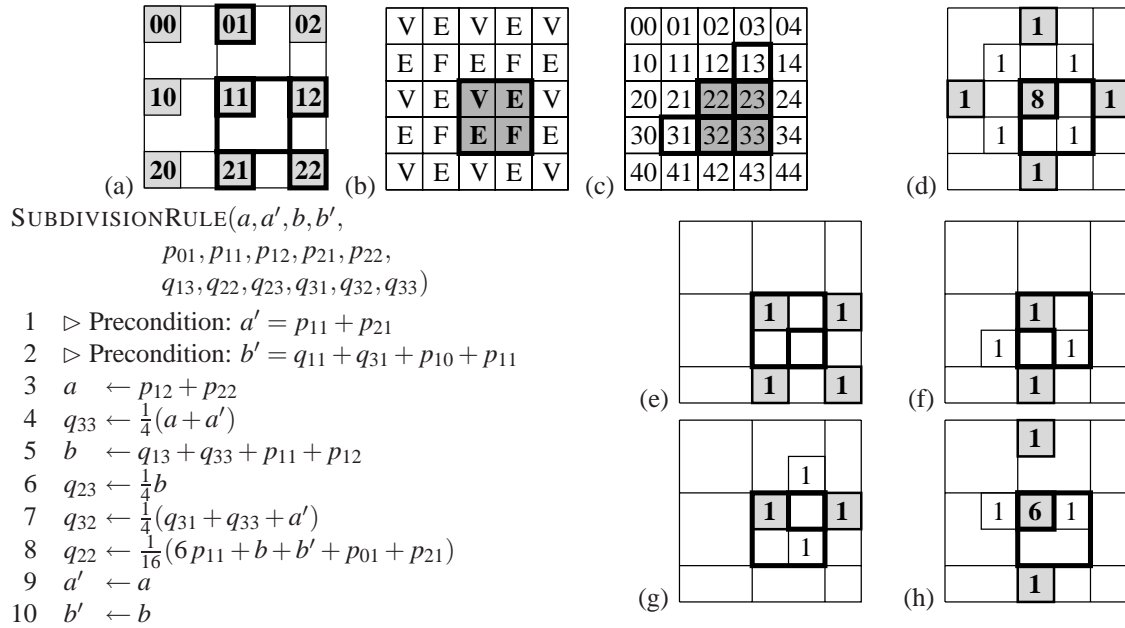


Figure 3.34: Subdivision rule for a 2×2 block and local point enumeration scheme. Given points $p_{0\dots2,0\dots2}$ from the previous level (a), the four points $q_{22}, q_{23}, q_{32}, q_{33}$ from the next level (c) are to be computed, which are a vertex, edge, edge, and face points (b). Blocks are computed row-wise from top left to bottom right, so points q_{13} and q_{31} are already available. The vertex rule (d) has a large stencil, but it can make use of the right and left edge points (h). Further operations are saved by using temporary variables $a' = p_{11} + p_{21}$ and $b' = q_{11} + q_{31} + p_{10} + p_{11}$. This reduces the operation count to $11/4$ vecadd and $5/4$ vecmul for creating one point of the refined grid.

variables to reduce the overall absolute cost. This cost is measured as the *operation count*, which is the number of *vecadd* and *vecmul* operations. Each of them consists of three of their floating-point counterparts.

The four points of a block are computed with 11 *vecadd* and 5 *vecmul* operations. This makes subdivision a quite inexpensive operation: **Only 2.75 *vecadd* and 1.25 *vecmul* operations are needed to obtain one subdivided point!**

Second pass: Progressively filling the Tessellation using the LIMITRULE. When the grid is refined new points can be added to the tessellation. This proceeds in an up-sampling manner: With $k_{\max} = 4$ the 9×9 arrays are filled after three rounds of refinement, as can be seen in 3.32 (2b-2d). This figure shows the relation between the grid, which grows in size, and the fixed-size tessellation. As only the missing points need to be added on every level, the *limit projection pattern* again induces a 2×2 tiling of the control grid (Fig. 3.32 upper row), and only 3 from 4 points in each block need to be projected. Note that this tiling is different from the one used in subdivision, the (already projected) vertex point is now in the lower right of a tile.

The computation is optimized based on the observation that the regular position and tangent stencils use weighted (1,4,1) sums. This applies to the position stencil as well, which is a (1,4,1) sum of (1,4,1) sums! In order to exploit this fact, three sets of temporary variables are maintained when processing the tiles in row-wise order (top to bottom, left to right): The column sums a_3 and a_4 will be the $\mathbf{a}_0, \mathbf{a}_1$ for the next tile (to the right), and the row sums b_3 and b_4 will take the roles of $\mathbf{b}_0, \mathbf{b}_1$ for the respective tile on the next row of tiles (to the bottom). The t_y -tangent of point 12 needs additionally row sums \mathbf{c}_0 and c_1 , and c_1 will be the \mathbf{c}_0 on the next tile row. Similarly, the t_x -tangent of point 21 needs additionally \mathbf{c}_2 and c_3 , and c_3 will take the role of \mathbf{c}_2 when the next tile (to the right) is processed.

When these values are available, the bold lines in procedure *LimitRule* can be omitted, and the operation count reduces to 24 *vecadd*, 12 *vecmul*, and 3 *unitnormal* operations for the projection of three points. The *unitnormal* operation $\text{unitnormal}(t_x, t_y) := \text{normalize}(t_x \times t_y)$ is a cross product followed by vector normalization, to obtain normal vectors of unit length. So in total, **the cost for the limit projection of one single point is 8 *vecadd*, 4 *vecmul*, and 1 *unitnormal*.** And what is the cost of a *unitnormal*? A cross product is equivalent to 1 *vecadd* + 2 *vecmul*, and vector normalization is a scalar product (equivalently $2/3$ *vecadd*, 2 *vecmul*), and an inverse square root (or a square root and a division). This makes *unitnormal* an expensive operation: $5/3$ *vecadd*, 4 *vecmul*, and an inverse square root.

LIMITRULE($a_{0..3}, b_{0..3}, c_{0..3}, p_{0..3,0..3}$)

```

1  a0 ← p00 + 4p10 + p20
2  a1 ← p01 + 4p11 + p21
3   $a_2 \leftarrow p_{02} + 4p_{12} + p_{22}$ 
4   $a_3 \leftarrow p_{03} + 4p_{13} + p_{23}$ 
5  b0 ← p00 + 4p01 + p02
6  b1 ← p10 + 4p11 + p12
7   $b_2 \leftarrow p_{20} + 4p_{21} + p_{22}$ 
8   $b_3 \leftarrow p_{30} + 4p_{31} + p_{32}$ 
9  c0 ← p01 + 4p02 + p03
10  $c_1 \leftarrow p_{21} + 4p_{22} + p_{23}$ 
11 c2 ← p10 + 4p20 + p30
12  $c_3 \leftarrow p_{12} + 4p_{22} + p_{32}$ 
13  $v_{11} \leftarrow \frac{1}{36}(a_0 + 4a_1 + a_2)$ 
14  $v_{12} \leftarrow \frac{1}{36}(a_1 + 4a_2 + a_3)$ 
15  $v_{21} \leftarrow \frac{1}{36}(b_1 + 4b_2 + b_3)$ 
16  $n_{11} \leftarrow \text{normalize}((a_2 - a_0) \times (b_0 - b_2))$ 
17  $n_{12} \leftarrow \text{normalize}((a_3 - a_1) \times (c_0 - c_1))$ 
18  $n_{21} \leftarrow \text{normalize}((c_3 - c_2) \times (b_1 - b_3))$ 

```

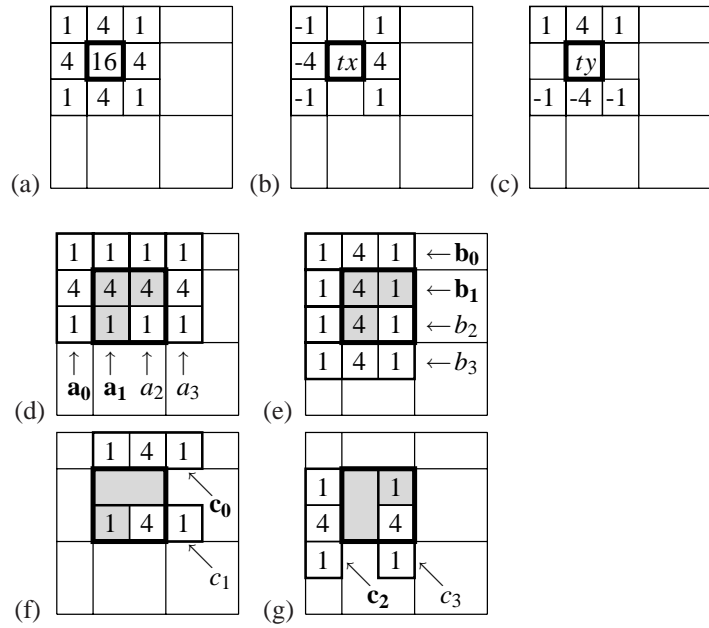


Figure 3.35: Limit projection rule for a 2×2 tile, with the same local enumeration scheme, but different tiling than before. The bottom left point of each tile was projected on the previous level. To the other 3 points, the position and tangent stencils must be applied, shown in (a)-(c) for the top left point. – The computation of the bold variables $\mathbf{a}_0, \mathbf{a}_1, \mathbf{b}_0, \mathbf{b}_1, \mathbf{c}_0, \mathbf{c}_2$ can be saved using temporary variables, as they overlap with previous calls to the same function. This reduces the operation count to 8 vecadd, 4 vecmul, and 1 unitnormal per point projection.

Program code and coding techniques. The block-based functions for subdivision and for limit point projection are very efficient: Only 2.75 vecadd and 1.25 vecmul are needed to obtain a point of the regular control mesh on the next refinement level, and only 8 vecadd, 4 vecmul, and 1 unitnormal to obtain a point of the tessellation. But optimization has always two aspects: Optimizing the algorithm is one part, but to optimize also the implementation another. The performance gains from adapting to a specific hardware architecture can be drastic. For the implementation of subdivision, however, only generic optimizations were realized that are recommended for any C++ program. – The algorithm is not yet adapted to any specific hardware architecture, this might be subject to future work.

The subdivision and limit projection were combined for all levels together in one large subdivision function. For a maximum execution speed it is a *straight line program* for the most part: Loops are unrolled and conditional branches are avoided as much as possible. All temporary 3D points are stored in one ‘static’ point array. Nested inline functions help to minimize function calls and parameter passing over the stack, and they keep the code nevertheless quite concise. A portion of the actual C++ code is shown in Fig. 3.36. It illustrates the coding of the subdivision pass from level 2 to 3 and the limit projection pass on level 3, as depicted in Figs. 3.33 (3a,3b), and 3.32 (c), respectively.

The truth is that not only one, but four subdivision functions exist. Every edge of the base mesh can be either smooth or sharp. Since every patch is incident to two base edges, this makes for four possible combinations. To minimize branching each combination has its own function. Only the combination (smooth,smooth) was presented here, the other ones are analogous. They use the B-spline curve subdivision from section 3.1.2 for left or top borders of the patch that are sharp.

A note on reciprocal square roots and SIMD extensions. The reciprocal square root is a remarkable function because it can be computed much faster directly than a square root followed by a division. The Intel IA-32 processor architecture for instance provides two instructions, FDIV and FSQRT, which both take 23 clock cycles in single precision and 38 cycles in double precision on a Pentium 4 ([Int99b], Table C-6). For comparison: The single precision FADD and FMUL instructions take 5 and 7 cycles (latency), and have a throughput of 1 and 2 cycles, due to the pipelined architecture.

The newer SIMD-extension on Intel processors (single instruction, multiple data) however provides a RSQRTPS instruction ([Int99a], Chapter 3.2) which computes an approximation to the inverse square roots of four single precision numbers at a time – in only 6 clock cycles! This is about as fast as the SIMD instructions ADDPS and MULPS that add and multiply four single-precision floats at a time in 4 and 6 cycles. The problem with the RSQRTPS instruction is just that it delivers only 12 bits of accuracy. To improve this Intel proposes one iteration of the Newton-Raphson method,


```

// * SUBDIVIDE LEVEL 2 TO GET LEVEL 3 *****

vecsum    (aP,  B02,B12);
subdivTopRule(aN,aP, B03,B13,C02,C03,C04);
subdivTopRule(aP,aN, B04,B14,C04,C05,C06);

vecsum    (aP,  B12,B22);
vecsum1111 (bP,  B11,B12,C02,C22);
subdivRule(aN,aP,bN,bP, B02,B12,B13,B22,B23, C04,C13,C14,C22,C23,C24);
subdivRule(aP,aN,bP,bN, B03,B13,B14,B23,B24, C06,C15,C16,C24,C25,C26);

subdivLeftRule(B20,B21,B30,B31, C20,C30,C40);
subdivRule(aN,aP,bN,bP, B11,B21,B22,B31,B32, C22,C31,C32,C40,C41,C42);
vecassign (aP,  aN);
vecassign (bP,  bN);
subdivLR1Rule(B12,B13,B22,B23,B24,B32,B33,
              C24,C26,C33,C34,C35,C36,C42,C43,C44,C46);

subdivLeftRule(B30,B31,B40,B41, C40,C50,C60);
subdivRule(aN,aP,bN,bP, B21,B31,B32,B41,B42, C42,C51,C52,C60,C61,C62);
vecassign (aP,  aN);
vecassign (bP,  bN);
subdivLR2Rule(B22,B32,B42, C53,C54,C62,C63,C64);

// * LIMIT POINTS LEVEL 3 *****

if (levelMax<3) {
  vecsum141 (bP, C01,C11,C21); // col 0
  limitTopRule(2, bP,bN,dP0,eP0, C01,C02,C03,C11,C12,C13,C21,C22,C23);
  limitTopRule(6, bN,bP,dP1,eP1, C03,C04,C05,C13,C14,C15,C23,C24,C25);

  limitLeftRule(18, C10,C11,C12,C20,C21,C22,C30,C31,C32);
  limitRule(20,22,38, aP,bP,aN,bN,dP0,eP0,dN0,eN0,
            C12,C13,C14,C21,C23,C24,C31,C32,C33,C34,C41,C42,C43);
  limitRule(24,26,42, aN,bN,aP,bP,dP1,eP1,dN1,eN1,
            C14,C15,C16,C23,C25,C26,C33,C34,C35,C36,C43,C44,C45);

  limitLeftRule(54, C30,C31,C32,C40,C41,C42,C50,C51,C52);
  limitRule(56,58,74, aP,bP,aN,bN,dN0,eN0,dP0,eP0,
            C32,C33,C34,C41,C43,C44,C51,C52,C53,C54,C61,C62,C63);
  limitRule(60,62,78, aN,bN,aP,bP,dN1,eN1,dP1,eP1,
            C34,C35,C36,C43,C45,C46,C53,C54,C55,C56,C63,C64,C65);
  if (level==3) { return; } // in case only level 3 was requested
}

```

Figure 3.36: Part of the patch tessellation function.

This excerpt of C++ code shows how subdivision level 3 is obtained from level 2. It demonstrates in particular the usage of the subdivision and limit rules for the individual blocks. Control points are indexed in a matrix fashion, with letters A, B, C, D for subdivision levels 1, 2, 3, and 4. The first part of the code performs the subdivision, the second part the limit point computation. They correspond directly to Fig. 3.33 (3a,3b). The limit code is executed conditionally, only if the points from level 3 are not yet in the tessellation. In case only level 3 was requested the function returns after the projection step.

Subdivision and projection are performed alternately, with only one or two if-statements per level. In case level 3 has already been projected before, and now level 4 is requested for refinement, the projection part is skipped and the computation proceeds with level 4 subdivision and projection. There is no caching of the (unprojected) subdivided vertices from previous calls: The memory overhead would be high (right column in Fig. 3.33), and the performance gain only minimal since subdivision only is not costly at all. Limit projection requires three times more operations than subdivision.

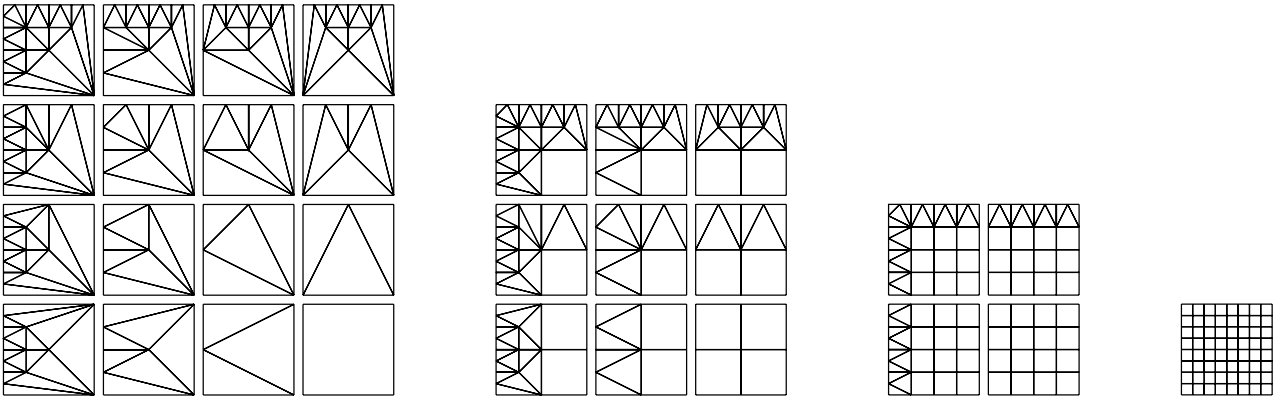


Figure 3.37: Adaptive refinement towards the patch borders. Cracks in the tessellation are avoided when low-resolution patches refine towards their higher resolution neighbors. The patch orientation is such that the vertex and face points in the TL and BR corners, so the patch meets its two neighbour along the left and top borders. The basic depth levels 1, 2, 3 and 4 are shown in the lower right, the refinements to its left and to the top. The level 1 (left) has the most possibilities for refinement. It can even be next to a level 4 patch.

which essentially doubles the number of significant bits in each iteration. Intel argues that it can be implemented very efficiently so that it takes less than another 6 cycles. In total, Intel claims, this gives a performance gain by a factor of 35, compared to executing four times FDIV and FSQRT ([Int98], Appendix A, “Newton-Raphson Method with the Reciprocal Instructions”).

In general, the performance gain from using SIMD can be a factor of 4, as four single precision or integer operands can be processed at a time. Optimal performance however, e.g., with Intel’s SSE2, can only be achieved with an appropriate data layout. This concerns the *AoS vs. SoA issue*: It makes a difference whether using an array of structs or a struct of arrays. In order to calculate four dot products $a \cdot b, c \cdot d, e \cdot f, g \cdot h$ from eight 3D vectors, the MULPS instruction can compute for instance the four products of the x -component in one step:

$$\text{MULPS} : (a_x, c_x, e_x, g_x) (b_x, d_x, f_x, h_x) \longrightarrow (a_x b_x, c_x d_x, e_x f_x, g_x h_x)$$

This requires, however, that the input vectors are arranged this way in memory, otherwise costly re-ordering, or *swizzling*, is necessary. Intel therefore proposes to store an array of 3D points in three arrays, one for the x , the y , and the z components of all vectors (SoA order). This can require substantial data structure re-design, especially with object-oriented approaches. Another possibility to achieve at least 75 percent of the peak performance, or a improvement of factor 3, is to use 3D vectors as SIMD operands:

$$\text{ADDPS} : (p_x, p_y, p_z, -) (q_x, q_y, q_z, -) \longrightarrow (p_x + q_x, p_y + q_y, p_z + q_z, -)$$

But not only is one component wasted in this case, the drawback is also that for optimal performance, the operands need to be aligned on 16 byte memory boundaries. A 3D vector of three single-precision floats needs only 12 bytes, so an array of memory aligned 3D points requires one third more memory. So although the performance gain for subdivision can be drastic, as e.g. demonstrated by Bolz and Schröder in [BS02b], substantial re-organization may be necessary to make optimal use of SIMD. The implementation of the presented scheme does not (yet) use SIMD to perform each vecadd and vecmul with only a single instruction. A speed-up by a factor of 3 could be achieved when using platform-specific code.

The multiplication with a power of two. Yet there is another remarkable low-level optimization, for which unfortunately no machine instructions seem to exist: The multiplication $a \cdot 2^b$ of a floating point number a by a power of 2. It could be replaced simply by adding b to the exponent of a . For single precision 32 bit IEEE floats, this is even just an 8 bit operation, since the exponent is stored in bits 23-30 (bit 30 is the sign of the exponent, bit 31 the sign of the number itself). This could only be efficiently exploited through a special machine instruction, because a multiplication is only 2 cycles slower than an addition anyways. The $a \cdot 2^b$ multiplication, though, could be even faster than the full float addition. For Catmull/Clark subdivision, this instruction would be quite beneficial: It is quite remarkable that the regular stencils contain almost exclusively multiplications by constant factors that are powers of 2 (see Figs. 3.34 and 3.35).

This fact is also one of the reasons for the striking computational stability of repeated recursive subdivision: Multiplications by a power of 2 can be computed *exactly* in the binary system. Another reason for its excellent numerical condition is the fact that the whole method is based on a system of repeated convex combinations. No subtractions are involved (except for the normals), and the numbers added are in the same orders of magnitude.

```

glEnableClientState (GL_VERTEX_ARRAY);
glEnableClientState (GL_NORMAL_ARRAY);
glVertexPointer (3, GL_FLOAT, 0, patch→points);
glNormalPointer (GL_FLOAT, 0, patch→normals);
glDrawElements (GL_TRIANGLE_STRIP, trilength [d][dLeft][dTop],
GL_UNSIGNED_SHORT, tristrip [d][dLeft][dTop]);
glDrawElements (GL_QUAD_STRIP, quadlength [d][dLeft][dTop],
GL_UNSIGNED_SHORT, quadstrip [d][dLeft][dTop]);

```

Figure 3.38: OpenGL calls to render a patch at a given resolution $(d, d_{\text{left}}, d_{\text{top}})$. The `glDrawElements` routine expects a primitive type, number of primitives, index data type, and a pointer to an array of indices. The indices in this array are relative to the vertex/normal pointers specified just before. The diagram shows the vertices used for resolution $(2,3,3)$: The indices of the triangle strip and the quad are pre-computed.

To further reduce the function call overhead, it is also possible to merge the quad strip into the triangle strip, when each quad is split into two triangles.

3.4.3 Adaptive Realtime Display

In order to eventually display a subdivision surface using graphics hardware, display primitives must be generated: On the lowest level everything boils down to triangles. To achieve output sensitivity, the tessellation must be adaptive. Adaptive display is complicated by the fact that when adjacent patches with different refinement depths are rendered using a *regular* tessellation, annoying cracks appear along their common border. A suitable tessellation scheme therefore has to take neighbour resolutions into account. So the rendering stage is organized in the following way:

- First, all base faces are assigned a refinement level d , also called the *face depth*, which is an integer ranging from -1 (not visible/backfacing, skip) to 4 (highest resolution).
- A degree n -face is partitioned into n patches, and every patch has common borders with two patches from neighbouring faces.
- When two patches differ in depth, the patch with the lower resolution refines towards the common border, to match the higher depth there.
- Assuming a patch orientation with the face point in the bottom right and the vertex point in the top left, the patch needs to be refined at most towards the left border or the top border.
- A suitable tessellation is thus chosen for each patch based on an integer triplet $(d, d_{\text{left}}, d_{\text{top}})$.

A face depth of 0 is a special case: If all faces have depth 0, the subdivision surface is identical to the base face, except that all vertices are projected to their limit position. The partition into quadrangular patches, and thus the the computed tessellation, is only used with depth 1 or higher. So in principle, $4 \cdot 4 \cdot 4 = 64$ combinations are possible for the $(d, d_{\text{left}}, d_{\text{top}})$ triplet. But all combinations where the face depth is equal or higher than the neighbour depth can be discarded as well: In this case it is the neighbour that has to refine. So for $d = 3$, only the cases where $(d_{\text{left}}, d_{\text{top}}) \in \{(3, 3), (3, 4), (4, 3), (4, 4)\}$ need to be considered. In summary only $4 \cdot 4 + 3 \cdot 3 + 2 \cdot 2 + 1 \cdot 1 = 30$ combinations are possible. They are shown in Fig. 3.37. All the triangle and quadrangle configurations are computed once in advance. Features are the following:

- Any difference in depth can be accomodated: A depth 1-face can be next to a depth 4-face.
- The arrays of such a depth 1-face must nevertheless be computed to depth 4, since it must refine to the border.
- Only high depth differences lead to long, thin triangles.

The original paper [Hav02b] has proposed a strategy for refinement towards the boundary where only the actually missing vertices are computed, rather than a complete level. In practice, however, it has proven very probable that a depth 1 face that is next to a depth 4 face is going to be shown in higher resolution itself at most a few frames later. To restrict the computation to whole levels also reduces the administrative costs.

To render a patch adaptively is extremely simple and effective using *vertex arrays*: As shown in the code example in Fig. 3.38, the pre-computed tessellation can be rendered with just a few OpenGL calls. OpenGL, as most low-level 3D APIs, offers the possibility to render indexed geometry. Thus, when the resolution of a face changes, just a different one from the 30 pre-computed index arrays needs to be selected according to the $(d, d_{\text{left}}, d_{\text{top}})$ triplet.

This approach regards subdivision surfaces merely as a specific rendering method for a base face: Instead of rendering the triangulated face the patches are rendered. In contrast to the original idea of recursively subdividing the control mesh, the base mesh is now left completely untouched. The great advantage is that **the resolution can be changed at no cost at all** once the tessellation itself is completely filled! Only this property makes it possible to adjust the resolution of all faces on a per-face-per-frame basis, which was one of the method's primary design objectives.



Figure 3.39: Adaptive display of subdivision surfaces.

3.4.4 Results and Discussion

The main result of the presented new computation scheme is that it reduces the computation cost by more than 40 percent compared to the previously published scheme [Hav02b]. To subdivide the quasi-regular grid the previous approach applied the vertex, edge, and face stencils in a straightforward way recursively. It used only a more efficient *final rule* for the combined computation of a limit surface point and its normal.

The previous approach was already more efficient than the basis function approach. In terms of operation counts, the new scheme is – to the best of the author’s knowledge – **the most efficient evaluation scheme to-date** for Catmull/Clark surfaces. The performance gain is illustrated in the following table. It sums up the operation count to obtain level 4 from the first subdivision:

Approach	#vecadd	#vecmul	#unitnormal
Refinement of vertex & face rings, levels 2-4	168	66	-
Grid refinement & limit projection, levels 2-4	876	410	81
Optimized recursive subdivision, levels 2-4	1044	476	81
Previously published recursive subdivision, levels 2-4	2071	638	81
Direct evaluation via basis functions	>3500	>3500	81

The third row is the sum of the first two rows with the two phases of subdivision and limit projection. The table shows that the operation count of subdivision *plus* projection is comparable to just about three or four times the unitnormal operations alone! The 81 unitnormals cost 135 vecadd, 324 vecmul, but also 81 inverse square roots (c.f. the note on inverse square roots in 3.4.2).

The operation costs from the optimized scheme have been counted directly from the code. Besides the subdivision and limit rules just presented, the numbers include also the overhead from the initialization of the top and left borders and the temporary variables. The unitnormal operations were counted separately, of course, because they cause the same cost with all approaches. The cost for setting up the vertex and face rings of level 1 was not counted either.

The cost of the *final rule*. The computation of the limit position and normal was combined into a single operation, the *final rule*, listed in Fig. 3.40. Points are indexed in matrix fashion (as in Fig. 3.35), with p_{11} in the upper left and center p_{22} . The final rule uses 16 vecadd, 5 vecmul, and 1 unitnormal. Alternatively, six weighted (1,4,1) sums can be used, and two vector subtractions for the tangents, which makes for 14 vecadd and 6 vecmul plus 1 unitnormal – useful only if a multiplication takes less than twice as long as an addition on the given hardware.

The cost of recursive subdivision. The operation counts for the basic Catmull/Clark subdivision rules are summarized in table 3.41. They are obtained by grouping vectors with the same weights together. To determine the total operation count for subdivision consider a single regular 4×4 patch. The k th subdivision of this patch contains $(2^k)^2$ quadrangles, which are defined by $(2^k + 1)^2$ vertices. Including the 1-neighbourhood, a grid of size $(2^k + 3)^2$ is needed to compute the next subdivision level. But how often did each of the rules have to be applied to produce these $(2^k + 3)^2$ points? Consider the following decomposition of this number:

$$(2^k + 3)^2 = ((2^{k-1} + 1) + (2^{k-1} + 2))^2 = (2^{k-1} + 1)^2 + 2 \cdot (2^{k-1} + 1)(2^{k-1} + 2) + (2^{k-1} + 2)^2$$

Careful inspection of Fig. 3.33 and, e.g., counting the red, green, and blue dots in Figs. 3.10 and 3.17 reveals that these three summands are just the number of times the different rules are applied in the regular case. They are listed in table 3.42.


```

FINALRULE( $n, p, p_{1..3, 1..3}$ )
1  $a \leftarrow p_{13} - p_{31}$ 
2  $b \leftarrow p_{11} - p_{33}$ 
3  $c \leftarrow p_{12} + p_{23} + p_{32} + p_{21}$ 
4  $d \leftarrow p_{11} + p_{13} + p_{31} + p_{33}$ 
5  $t_x \leftarrow a - b + 4(p_{23} - p_{21})$ 
6  $t_y \leftarrow a + b + 4(p_{12} - p_{32})$ 
7  $n \leftarrow \text{normalize}(t_x \times t_y)$ 
8  $p \leftarrow \frac{1}{36} (16 p_{22} + 4c + d)$ 
9 return  $n, p$ 
    
```

Figure 3.40: The final rule. Combined computation of limit position p and normal n from [Hav02b].

	vecadd	vecmul	unitnormal
vertex rule	8	2	
face rule	3	1	
edge rule	3	1	
<i>final rule</i>	16	5	1

Figure 3.41: Operation counts for regular stencils.

vertex rule: $(2^{k-1} + 1)^2$ times
 face rule: $(2^{k-1} + 2)^2$ times
 edge rule: $2(2^{k-1} + 1)(2^{k-1} + 2)$ times
final rule: $2^{k-1}(3 \cdot 2^{k-1} + 2)$ times

Figure 3.42: Number of times each rule must be applied to produce level k from level $k - 1$.

k	Grid	Tessel.	face	edge	vertex	final	vecadd	vecmul	unitnormal
0	4×4	2×2				4	64	20	4
1	5×5	3×3	9	12	4	5	239	74	9
2	7×7	5×5	16	24	9	16	687	212	25
3	11×11	9×9	36	60	25	56	2071	638	81
4	19×19	17×17	100	180	81	208	6887	2120	289

Figure 3.43: Number of rule applications on each of several levels of subdivision. The operation counts are accumulated: To produce a tessellation with 9×9 points, in total 2071 vecadd, 638 vecmul and 81 unitnormal operations are performed.

For the number of times the the final rule is applied in table 3.42 the usage of the upsampling scheme from Fig. 3.32 is assumed again. The $(2^{k-1} + 1)^2$ vertices from the previous level are already projected on the limit surface, and only the *additional* points need to be projected on every level. The number of remaining applications of the final rule is therefore $(2^k + 1)^2 - (2^{k-1} + 1)^2$. Concrete numbers for the resulting application counts for all four rules for the first four levels are given in table 3.43, along with the respective accumulated numbers of elementary operations. These absolute numbers show how important it is to optimize the final rule: The final rule is applied asymptotically more often than any other rule. Already for $k = 4$ this is the case (bottom line in table 3.42).

It is interesting that the average number of computations needed to produce one point in a tessellation is asymptotically constant. The higher the level of refinement the higher the cost, but also the larger the number of points in the tessellation. So what is the sum of vecadd and vecmul operations needed for producing a single point in a subdivision grid with $(2^k + 1)^2$ points when k approaches infinity? This is the total number of rule applications across all levels, divided by the number of tessellation points.

The vertex rule for instance is applied $(2^{k-1} + 1)^2 = 2^{2k-2} + 2^k + 1$ times, each time issuing 8 vecadd and 2 vecmul. The asymptotically dominant term is 2^{2k-2} . It turns out that the edge and face rules have the same dominating term 2^{2k-2} . To sum the rule applications over all levels a simple fact comes in handy: it is $\sum_{i=0}^{n-1} 2^{2i} = \frac{1}{3}(2^{2n} - 1)$. Taking into account that the edge rule is applied twice as often, the number of vecadd and vecmul per (unprojected) point on the refined grid with straightforward recursive subdivision is:

$$\begin{aligned}
 \#\text{vecadd}_{\text{lim}} &= \lim_{k \rightarrow \infty} \frac{1}{3} \frac{2^{2k} - 1}{(2^k + 1)^2} \cdot (8 + 3 + 2 \cdot 3) = \frac{1}{3} \cdot 1 \cdot 17 = \frac{17}{3} \\
 \#\text{vecmul}_{\text{lim}} &= \lim_{k \rightarrow \infty} \frac{1}{3} \frac{2^{2k} - 1}{(2^k + 1)^2} \cdot (2 + 1 + 2 \cdot 1) = \frac{1}{3} \cdot 1 \cdot 5 = \frac{5}{3}
 \end{aligned}$$

This means that in the limit, in fact for $n \geq 6$, less than 6 vecadd, 5 vecmul operations are needed to produce a vertex of the tessellation. To then project the vertex on the limit surface and to compute its normal using the final rule, additional 16 vecadd, 5 vecmul, and 1 unitnormal have to be performed. So in total, around 22 vecadd, 10 vecmul, and 1 unitnormal operations are needed for each point of a regular tessellation, using this straightforward implementation. These constants are of course smaller with the new optimized scheme. Subdivision takes asymptotically $11/3$ vecadd and $5/3$ vecmul, and limit projection another 8 vecadd, 4 vecmul, and 1 unitnormal.

This makes asymptotically 12 vecadd, 6 vecmul and 1 unitnormal per point/normal pair, a saving of more than 40%.

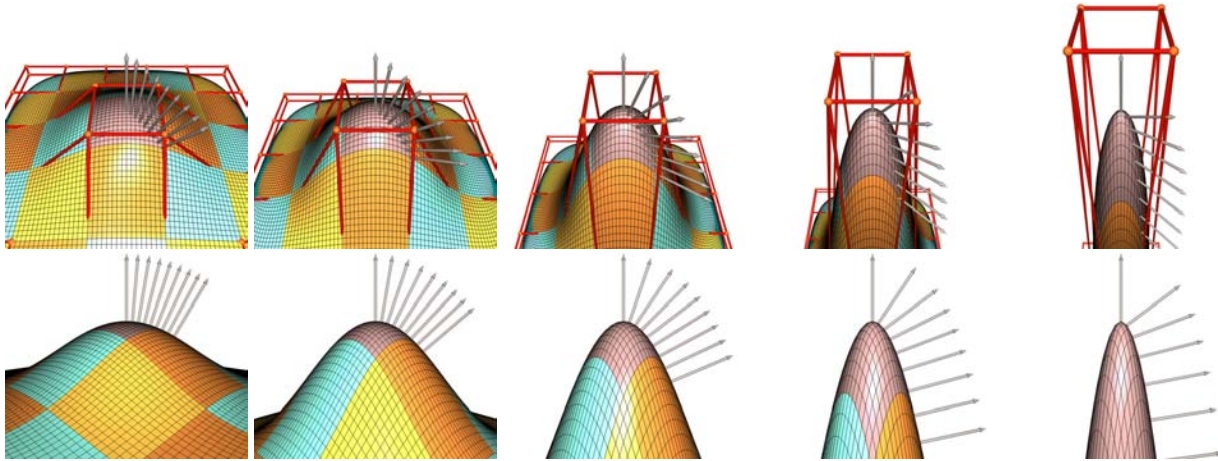


Figure 3.44: Extreme bending of a subdivision patch. Top: The grid faces are 1×1 units, the vertical displacements are 1, 2, 4, 8, and 16 units (1a-e). Bottom: Tessellation normals along the patch diagonal, from face to vertex point.

3.4.5 Remarks on Tessellation Quality and Accuracy Issues concerning $k_{\max} = 4$

The proposed method uses a fixed maximum refinement level $k_{\max} = 4$. The practical justification is that this permits more aggressive optimization, for instance static pre-allocated caches, pre-computed strip indices, unrolling of all loops, and to avoid any recursive function calls. But of course the question arises whether any fixed maximum refinement level can ever be sufficient, and in particular, whether $k_{\max} = 4$ is sufficient. Is this a serious limitation? – After examining this question two possibilities are discussed to extend the algorithm to arbitrary depth.

When is $k_{\max} = 4$ sufficient? The question of adaptivity can as well be posed the other way around: Assuming the highest resolution i is limited, what are the implications for modeling? In which occasions, and with which kind of control meshes will this limitation lead to noticeable artifacts? – Following the idea of output sensitivity, i.e., screenspace error, basically two criteria are important to steer the further refinement:

- **Projected size:** A given quad covers too many pixels
- **Curvature:** The piece-wise linear approximation is too far away from the true surface.

A violation of the projected size criterion becomes especially noticeable as *shading artifacts*, as demonstrated in Fig. 3.45. Gouraud shading interpolates the color values at the vertices only linearly, so it can not faithfully reproduce highlights with a too coarse sampling, especially with highly reflective materials. But a denser surface sampling is not the only remedy: With today’s programmable graphics hardware, especially with the advent of *per-pixel lighting* [Nvi04], much more powerful and general shading models have now become available, for instance even Phong shading.

h/w	$\sum_{i=0}^7 \alpha_i$	α_0
1.0	30.51	4.83
2.0	49.68	9.60
4.0	67.01	18.68
8.0	78.02	34.07
16.0	83.95	53.52
32.0	86.96	69.71
64.0	88.48	79.53
128.0	89.24	84.72

Table 3.4: Normal variation over the patch vs. variation over the first quad only

summarized in Table 3.4, reveal that in case of, e.g., a box that is 16 times as long as it is wide, the level 4 tessellation is definitely insufficient (c.f. Fig. 3.44, 2e): The angle between the face and vertex point normals is 83.95 degrees while α_0 is 53.52 degrees! This means that most of the variation of the whole patch actually occurs within the quads around the face point. The normals no longer vary smoothly over the patch.

The curvature criterion is also related to intrinsic properties of the surface. A limited sampling density will always fail when the surface curvature can be unlimited. But the curvature of the surface is determined by the ‘curvature’ of the control mesh. It is large when dihedral angles are large, the angles between the surface normals of neighbouring quads. But the dihedral angle between base faces can at most approach 180 degrees – assuming that the control mesh does not intersect itself. Worst cases are very long spikes, i.e., vertices or faces that are displaced very much in one direction. In order to better understand the worst cases, two experiments have been carried out.

The first experiment is shown in Fig. 3.44, where one face of a regular tessellation is displaced in vertical direction. The angular variation over the tessellation is measured: The tessellation of a base quad contains 16×16 quads on level 4. Consequently there are seven tessellation points along the patch diagonal between the limit face point and a limit vertex point. In the experiment the normal variation over the eight points was related to the angle α_0 between the normals at the limit face point and its diagonal neighbour. The results,

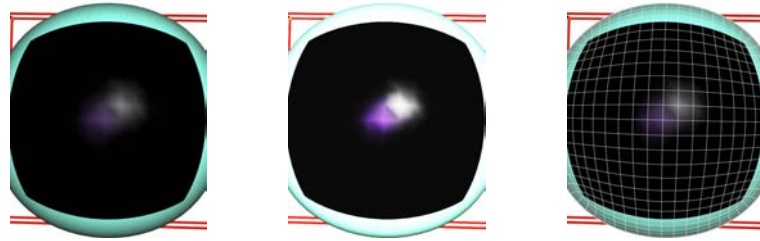


Figure 3.45: Shading artifacts from insufficient sampling. (a): Gouraud shaded dark but highly reflective surface. (b): Same as (a) with enhanced contrast and brightness. (c): The projected size criterion is violated.

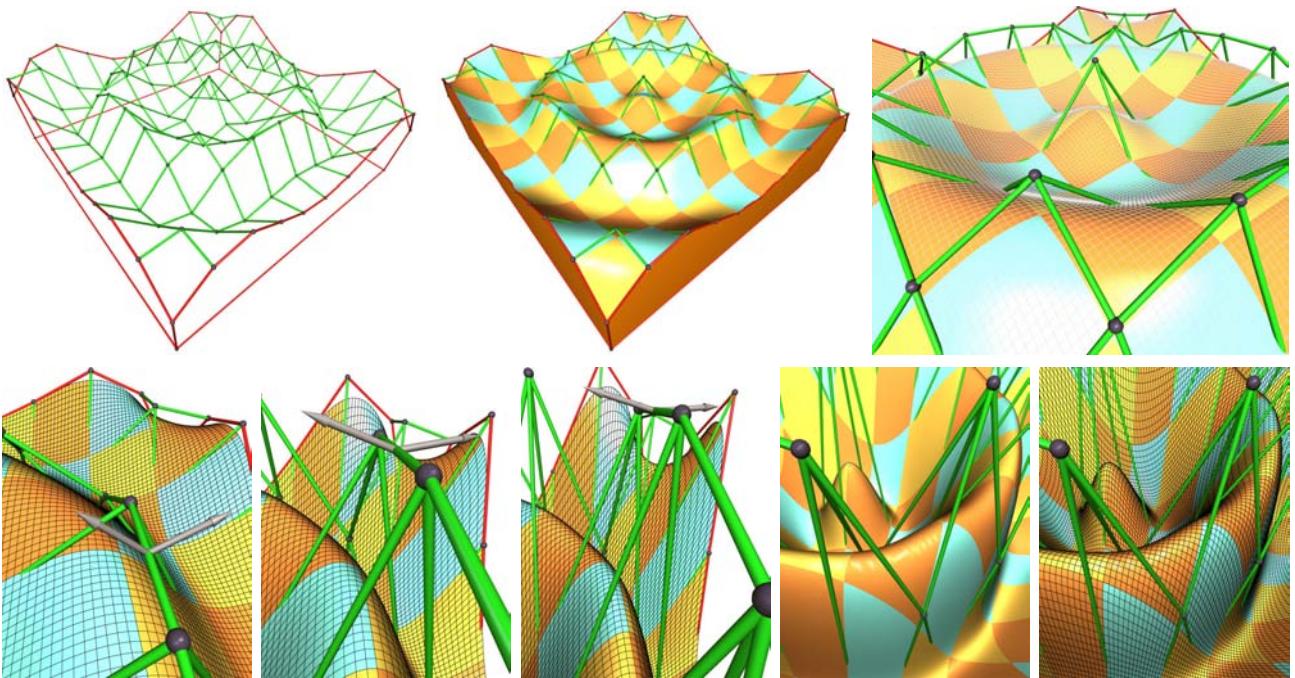


Figure 3.46: Subdivision surfaces approximating a sinoidal wave. Top: control mesh, subdivision surface, and level 4 tessellation. Bottom: Dihedral angles of 111 (a), 149 (b), and 160 degrees (c), and highly non-planar quads (c,d).

The second experiment is a sinusoidal wave, inappropriately represented with a coarse rectangular regular grid as control mesh, shown in Fig. 3.46. The grid CVs are displaced in vertical direction only, and the factor of displacement was varied. This experiment shows that also large dihedral angles can be accommodated: Even the 149 degrees between the base face normals in image 3.46 (2b) are sufficiently smoothed out by the recursive refinement.

Distinct shading artifacts appear in this model only in regions with a great distortion *within* a base face. The sinusoidal wave has a circular shape, and when the vertical grid displacement is large, the faces along the grid diagonal (the line of sight in 3.46 (1c)) deviate very much from a plane: The normals of opposite base quad vertices can nearly point in opposite directions, as in 3.46 (2d-2e). The quads in the tessellation inherit this bending, and the result is that Gouraud interpolation between two darker and two brighter vertices on opposite sides of a tessellation quad produces shading artifacts.

The third area where a limited sampling density is the problem that can lead to annoying artifacts was mentioned already: vertices with very high valences and high-degree base faces. The resulting artifacts can nicely be seen in Fig. 3.25, (1d) and (1e), where the tessellation quads around the vertex and face points are unproportionally large.

The bottom line of these experiments is that to limit the recursive refinement to level 4 is sufficient in most practical cases. The surface sampling was shown to be inappropriate in certain worst cases. It is not very probable, however, that such degenerate configurations are likely to be introduced intentionally by a designer. It can be expected that any artist with a minimum of experience most likely seeks to avoid bending single faces more than 60 or 70 degrees, dihedral angles over 150 degrees, and to use very thin, long, boxes to define a smooth surface.

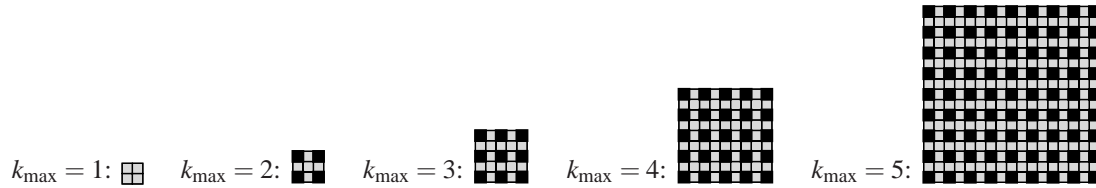


Figure 3.47: Alternative technique: Multiple fixed-size tessellations with copying prior to refinement.

Extensions to the framework for higher levels of refinement. There are basically two ways to extend the framework. The first is to use multiple instantiations of the technique, but each with a different k_{\max} . Instead of a fixed size tessellation there is a pool of patches of different sizes, such as shown in Fig. 3.47. Refinement is done via up-sampling just as before. But in order to refine, e.g., the 5×5 tessellation from level 3, it is copied to level 4 in a doubly spaced manner, filling only the black squares. The grey vertices remain to be filled in on level 4. This technique permits to display a few patches in very high refinement, e.g., in depth 5, 6, or even 7, without having to allocate the same amount of memory for all other patches as well: When using $k_{\max} = 4$ exclusively, 100K patches are allocated for a control mesh with 25K smooth quad base faces. This consumes of 194 MB of memory, instead of just 21.6 MB needed for, e.g., the level 2 tessellation. The alternative technique trades time for space, and the $(2^k + 1)^2$ copy operations may be a considerable time overhead, especially with higher levels of refinement. This technique is therefore advantageous in cases where the patch resolutions are greatly varying, and the resolution of most patches remains constant over some time.

The second possibility to extend the framework is to retain a fixed k_{\max} , but to use a hierarchy of patches: When higher levels than k_{\max} are needed, the patch is decomposed into sub-patches. Instead of splitting one quad into four sub-quads, a 3×3 patch is split on demand into nine 3×3 sub-patches, or a 5×5 patch into twenty-five 5×5 sub-patches. This reduces the administrative overhead and also the costs for hierarchy traversal. The drawback is that pre-computing the triangle strip indices gets more complicated, since many more cases need to be considered.

Next higher layer: The meshes to control subdivision. This chapter has presented the essential prerequisites to use subdivision surfaces for interactive free-form shape design: A very fast, refineable tessellation on-the-fly, and adaptive multi-resolution rendering. But to start with subdivision, first some form of control mesh is needed; in particular, a data structure for it.

Interesting shapes are not only made of free-form parts. Subdivision surfaces open the interesting option to represent polygonal and free-form surfaces with the same data structure. It is neither obvious nor trivial, though, how to organize the control mesh so that it can represent both types of surfaces efficiently. Furthermore, the surface has to allow for *selective updates*: When interactively changing a small part of a large control mesh, only the affected parts of the tessellation need to be re-generated, and ideally at interactive rates. – How this can be achieved is the subject of the next chapter.

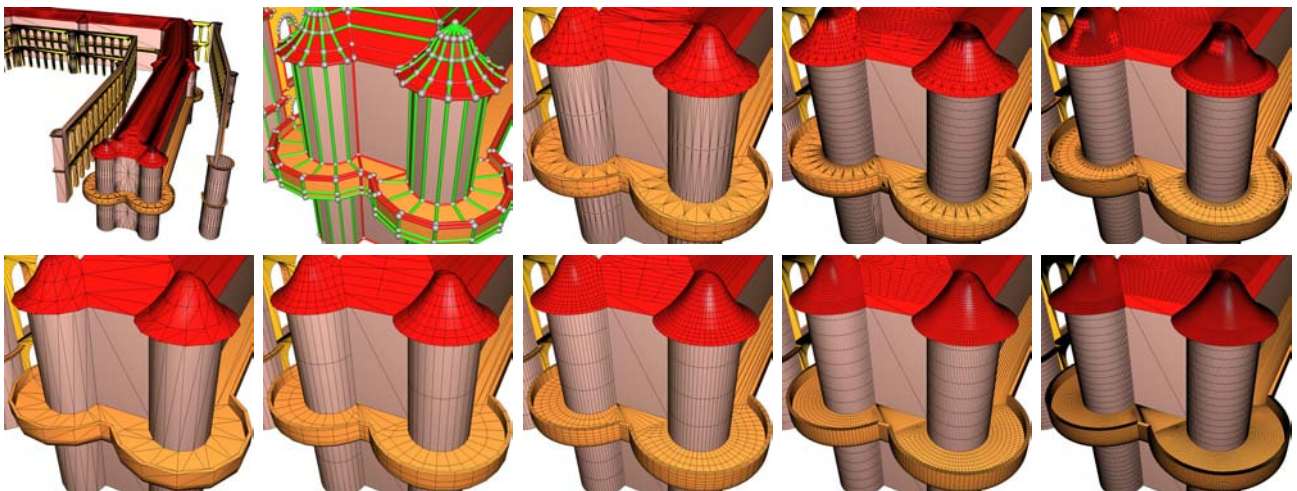


Figure 3.48: Adaptive display of large subdivision surfaces. For large control meshes a gradual LOD distribution is essential to maintain interactivity (1c-1e). A uniform tessellation, with smooth base faces 0-4 times subdivided (2a-e), is not an option. (1b): But where do the control meshes come from to start with?

Chapter 4

Practical Meshes

This chapter introduces the indispensable infrastructure for storing and manipulating polyhedral shapes in a digital computer, mesh data structures. It attempts to pick up and operationalize the abstract concepts from chapter 2. Consequently, this chapter is also somewhat more implementation oriented. In the context of the bottom-up approach pursued in this thesis, the mesh layer is the next higher layer above the subdivision surfaces from the previous chapter. The mesh layer is not a single layer, though, as it is in fact divided into three parts, or sub-layers:

- **B-rep Meshes** are a *container data structure* to represent the connectivity of a mesh. They are abstract, and not at all restricted to being used for 3D shapes. They can be thought of as a generic data structure for arbitrary locally planar graphs. B-reps can be augmented (*instantiated*) with any kind of custom data attached to vertices, edges, and faces.
- **Combined B-reps (or “cB-reps”)** are an instantiation of the general B-reps. Their set of custom data, their *trait*, includes 3D points for vertices, triangulations for polygonal faces, and subdivision surfaces for the curved parts of the surface. Thus, combined B-reps *bridge the gap* between polygonal and free-form shapes! They provide methods to find the visible faces (view cone culling) and to determine a suitable surface resolution on-the-fly (depth assignment).
- **Progressive Combined B-reps (“pcB-reps”)** finally are used to actually change and modify a combined B-rep mesh. They provide a concise API with only thirteen well-specified methods: The five Euler operators, their inverse operators, and for the manipulation of vertex, edge, and face attributes (3D position, sharpness, material). Furthermore, all executed mesh operations are *logged*, so that pcB-reps provide a complete undo/redo mechanism. It is even fast enough to serve as level-of-detail technique to temporarily remove un-needed detail at runtime.

In order to explain the relations between the layers, to motivate the design decisions, and to introduce the concepts used herein, the chapter begins with an introduction on the design options for mesh data structures. For many professionals ‘mesh’ means just ‘triangle mesh’. In order to explain why those were *not* used in this thesis they are discussed first.

A mesh survival guide. Meshes are a very particular object of study: A mystery and a myth to novices, but also a field for religious battles and quarrels for those who know. The truth is probably in between: No years of theory are required, but a sound background in algebraic topology helps tremendously – and even without that, a few basic facts already facilitate the understanding of mesh data structures a lot. Also true is that meshes come in a great variety and different flavors, because they are used for the most different purposes. Unfortunately it can not be expected that there is a single data structure that fits for all purposes in all domains. Especially with real-time applications, a penalty for using over-generalized data structures is unacceptable; only to load and display a 3D model does not even require a mesh. But on the other hand it is equally annoying to find out later in the application development process that a specific algorithm only performs well when the neighbourhood of an entity can be efficiently determined.

So is any attempt to find a unified view on meshes void from the start, will no single API ever meet all requirements? The challenge is to find a level of abstraction that is high enough to permit flexibility on the implementation side, but is also not too high as to be inefficient in giving access to the manipulation of low-level entities. Great unification efforts have been made by a number of famous initiatives, just to name CGAL [Vel99, SVY99], OpenCascade [Opea], or, more recently, OpenMesh [Bot, BSBI02]. Yet none of them has achieved general acceptance today: A query on sourceforge alone for ‘mesh 3d’ lists two dozen open source software packages with meshes in them [Sou]. Not to mention that probably every computer graphics group, and every 3D software company, has its own collection of mesh data structures, highly optimized for the respective purposes.

All design decisions imply certain trade-offs, and it is a laborious and sometimes repetitive task to find out what level of flexibility is required by an application that, e.g., a student has in mind. Some of these issues are discussed in the first section, which is supposed to serve as a cookbook with recipes hopefully valid for meshes of different flavors.

4.1 Triangle Meshes

When arbitrary shapes are to be represented, the first choice are always *triangle meshes*, or, following the nomenclature from chapter 2, simplicial complexes. This is due to

- their perceived simplicity, as three points always form a plane,
- their compatibility with graphics hardware,
- few limitations and validity constraints to cope with,
- availability of simplification and continuous adaptive level-of-detail,
- and their broad support in the research literature, with a wealth of sophisticated
- algorithms for mesh editing, extraction, re-sampling, segmentation, and parametrization, and many more.

It is probably fair to say that triangle meshes are the *smallest common denominator* for representing surfaces in 3D. The reason is that they can be exported from practically all 3D applications, irrespective of the respective internal surface representations: Any surface in 3D can be approximated by a triangulated, simplicial surface, and this even to arbitrary precision. And once exported, all the machinery developed for mesh processing can be applied.

There are some interesting direct consequences of the Euler-Poincaré equation (Theorem 2.27) for manifold triangle meshes. Triangles do not have rings, and it is reasonable to assume that in a practical triangle mesh the number of shells, topological holes, and boundary components is small compared to the number of vertices, edges, and faces. Furthermore, every triangle has three edges, and every edge belongs to two triangles. This can also be imagined as every edge being split in two halves, each half belonging to one of the two adjacent faces.

Theorem 4.1 (Number of entities in a triangle mesh)

Let v , e , and f be the number of vertices, edges, and faces in a manifold triangle mesh, where the number of shells s , topological holes h , and boundary components b is relatively small. Let s_v, s_e, s_f be the size of the vertex, edge, and face data structures. The number of ‘edge halves’ in a triangle mesh is $2e \approx 3f$, which leads to the following approximations:

- $v + f \approx e$
- $f \approx 2v$, in average there are twice as many triangles as vertices
- $e \approx 3v$, in average there are three times more edges than vertices
- $\text{memorysize}(V, E, F) = v \cdot s_v + e \cdot s_e + f \cdot s_f \approx v \cdot (s_v + 3s_e + 2s_f) \approx f \cdot (\frac{1}{2}s_v + \frac{3}{2}s_e + s_f)$

The first approximation holds for manifold meshes in general because it follows from $v - e + f = 2(s - h) - b$ when the right-hand side is close to zero. And since $2e \approx 3f$, the second approximation follows, because $2(v + f) \approx 2e \approx 3f$. And the third approximation is a consequence of the first two ones. – Although this may come as a surprise at first, it becomes plausible when considering for example regular grids, see Fig. 4.1. The implication for data structure design is that, e.g., one byte of memory saved with edge data is worth three bytes of memory saved with vertex data.

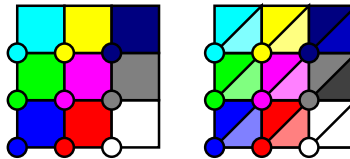


Figure 4.1: Regular grids of quadrangles and triangles. Every vertex of a regular quadrangle grid, which is a tiling of the plane, corresponds to one quadrangle: For quad meshes holds in essence $f \approx v$, and the average vertex valence is 4. In triangle meshes, the average valence is 6, so every vertex has twice as many edges as every face. Consequently, there must be twice as many faces: $f \approx 2v$.

4.1.1 “My First Triangle Mesh”: The Shared Vertex Data Structure

The first question is which programming language to use for implementing meshes. Mesh entities, i.e., vertices, edges, and faces, are obvious candidates for *objects* in the object oriented sense. On the other hand, especially for the performance required for real-time rendering, the C programming language is a good choice, due to its relative closeness to hardware and low-level drivers: The OpenGL API, for instance, is in C. So C++, the object-oriented extension of C, appears to be a feasible choice and a good compromise [ES90].

A first attempt to create a data structure for triangle meshes in C++ might look like in Fig. 4.2: A mesh consists only of two (fixed-size) arrays, one for vertices, and one for faces. A vertex contains just its 3D position and a reference to one of the triangles it is incident to. A face contains references to three vertices, and to three neighbour faces. Edges are not

```

struct Vec3f { ... float x,y,z; };

struct VertexSVNstatic
{
    int oneFace;
    Vec3f position;
};

struct FaceSVNstatic
{
    int vertex [3];
    int neighbour [3];
};

struct MeshSVNstatic
{
    int vertexN, faceN;
    VertexSVNstatic vertices[100];
    FaceSVNstatic faces [100];
};

void render(MeshSVNstatic& mesh)
{
    FaceSVNstatic* face = mesh.faces;
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(3, GL_FLOAT, sizeof(VertexSVNstatic),
                  &mesh.vertices[0].position.x);
    for(int i=0; i<mesh.faceN; ++i, ++face) {
        glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_INT,
                      face->vertex);
    }
    glDisableClientState(GL_VERTEX_ARRAY);
}

```

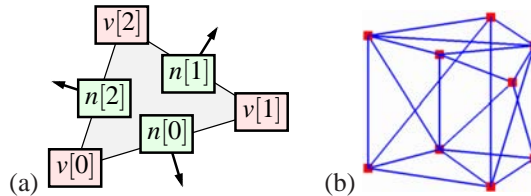


Figure 4.2: Shared vertex triangle mesh with neighbourhood information, static version. Each face contains three pointers to neighbour faces, each vertex has a face pointer. Both together make up the connectivity of the mesh. Neighbour and vertex references must be consistent with the orientation (a). The mesh can be readily rendered even without any connectivity information, (b) is a wireframe output captured from OpenGL.

explicitly represented, and references are realized by indices. This mesh representation of indexed triangle sets is called *shared vertex data structure* because triangles share their vertices. A vertex does not have to be copied to be part of more than one triangle. The data structure in Fig. 4.2 can additionally store the connectivity, so, strictly spoken, it is a ‘shared vertex mesh with neighbourhood information’.

The C++ class `Vec3f` is equally used for points and vectors, to keep things simple. In addition to the three floats from the code example it provides the usual methods such as `dot`, `cross`, `normalize`. – On 32 bit architectures, pointers as well as integers and single precision floats all require 4 bytes of memory, the *ID size unit* of 32 bit. Following theorem 4.1, the size of the shared vertex data structure is $s_v = 1 + 3 \text{ units} = 16 \text{ bytes}$, $s_e = 0$, and $s_f = 2 \cdot 3 \text{ units} = 24 \text{ bytes}$.

A shared vertex mesh with v vertices requires $(s_v + 2s_f) = 16 + 48 = 64 \text{ bytes}$ per vertex and, equivalently, a mesh with f triangles requires 32 bytes per triangle.

Loading and rendering a mesh. The shared vertex data structure can represent an indexed face set, as presented in section 2.3.2, that is loaded from an `.obj` or `VRML` file (Fig. 2.23), provided

- the file contains exclusively triangles, so it is actually an indexed triangle set, a ‘*triangle soup*’, and
- the arrays for vertices and faces are of sufficient size.

Indexed triangle sets provide no neighbourhood information. By default all neighbour indices are set to *invalid*, i.e., to the special value -1 . The same value is reserved for faces that have no neighbour face, i.e., faces at the border. This rule is consistent in that all triangles are isolated in the beginning.

A mesh without explicit connectivity is not completely useless. The shared vertex data structure permits, e.g., to deform a mesh in arbitrary ways: When the position of a vertex is changed, all incident triangles are immediately and consistently updated, since they directly refer to this vertex. Second, the mesh can be readily rendered using OpenGL. The code to render an indexed face set was shown in Fig. 2.24 from chapter 2. Basically the same is possible for shared vertex meshes, as the code example to the right of Fig. 4.2 suggests. The same object as in chapter 2 is shown in Fig. 4.2 (b) as a wireframe triangle mesh. This image is an authentic screen shot of the output from the render routine in the code example.

Connectivity definition. Since neighbourhood information can be stored a number of in different ways, it is necessary to specify conventions how to use the available indices. This helps to tell whether a given index assignment is consistent.

For a face, vertex and neighbour indices shall follow the convention depicted in the diagram in Fig. 4.2 (a): neighbour index i refers to the unique neighbouring triangle that is also incident to the vertex referred to by vertex indices i and $i + 1$, where $0 \leq i \leq 2$. In case an edge belongs to a border, the respective neighbour index remains -1 . In triangle meshes, closed border loops cannot be treated like faces, as it was proposed in chapter 2 (e.g., in theorem 2.23), simply because in general borders have degree > 3 . Triangle indices, such as $i + 1$, are understood as modulo 3, since the sequence is cyclic

```

void collectFaceNeighbours (MeshSVNstatic& mesh)
{
    typedef pair<int, int> Edge;
    typedef pair<Edge, int> EdgeFacePair;

    map<Edge, int> edges;
    Face* face;
    int k;

    for (face = mesh.faces, k=0; k<mesh.faceN; ++face, ++k) {
        edges.insert (EdgeFacePair (Edge (face->vertex [0], face->vertex [1]), k));
        edges.insert (EdgeFacePair (Edge (face->vertex [1], face->vertex [2]), k));
        edges.insert (EdgeFacePair (Edge (face->vertex [2], face->vertex [0]), k));
    }

    map<Edge, int>::iterator edge10, edge21, edge02;

    for (face = mesh.faces, k=0; k<mesh.faceN; ++face, ++k) {
        edge10 = edges.find (Edge (face->vertex [1], face->vertex [0]));
        edge21 = edges.find (Edge (face->vertex [2], face->vertex [1]));
        edge02 = edges.find (Edge (face->vertex [0], face->vertex [2]));
        face->neighbour [0] = (edge10 == edges.end ()) ? -1 : (*edge10).second;
        face->neighbour [1] = (edge21 == edges.end ()) ? -1 : (*edge21).second;
        face->neighbour [2] = (edge02 == edges.end ()) ? -1 : (*edge02).second;
    }
}

```

Figure 4.3: Gathering connectivity information. The function `collectFaceNeighbours` builds up on the shared vertex mesh implementation from Fig. 4.2. It uses the `map` template class from the STL as a dictionary containing (key,value) pairs with an integer pair as the key and one integer as value. Note that the edge insertion in the first pass does not check whether a key, i.e., an edge, already exists in the map.

and only its order counts. It is not important which of the vertices is referred to by which index, since all three orders (v_a, v_b, v_c) , (v_b, v_c, v_a) , and (v_c, v_a, v_b) are equivalent.

To distinguish between references and entities the notations $v[i]$ and $n[i]$, or $f.v[i]$ and $f.n[i]$, respectively, are used for vertex and neighbour references. The actual entities are denoted v_a, v_b, v_c or f_0, f_1, f_2 . – The cyclic order also defines the surface orientation; $n[0], n[1]$ and $n[2]$ are arranged in *CCW orientation*, and this shall denote the frontside of the triangle. A pair of adjacent triangles is *oriented consistently* if and only if both are incident to two vertices v_a and v_b , and one of them contains edge (v_a, v_b) , and the other contains (v_b, v_a) . Note that this gives a criterion to determine the consistency of the mesh, in the sense of the mesh consistency theorem (Theorem 2.24 from chapter 2).

Concerning mesh consistency, note that a vertex can store only a single reference to an adjacent face, i.e., to only one edge cycle. This means that complex vertices (as listed in the ‘mesh pitfalls’ table in Fig. 2.28) cannot be represented, since it would require several links to incident faces from different cycles. Edges are not explicitly represented in this data structure. This is another important design decision.

Gathering connectivity information Neighbourhood information is essential for almost every method to process a mesh. Some of the applications were mentioned in the beginning of this section (4.1): Mesh editing, simplification, reparametrization, and resampling, all need to ‘crawl’ over the surface, which is not possible without connectivity information. – So far, the neighbour indices of the mesh are still uninitialized. A simple 2-pass algorithm can determine the suitable neighbour indices by setting up and examining a search structure for directed edges.

- For each triangle f_k , $k = 1 \dots n$, with vertex indices (v_0, v_1, v_2) , enter the directed edges $(v_0, v_1) \rightarrow k$, $(v_1, v_2) \rightarrow k$, $(v_2, v_0) \rightarrow k$ into the search structure.
- For each triangle f_k , $k = 1 \dots n$, look up the reversed directed edges $(v_1, v_0) \rightarrow k_{10}$, $(v_2, v_1) \rightarrow k_{21}$, $(v_0, v_2) \rightarrow k_{02}$. The neighbour entries of triangle f_k are then $(n_0, n_1, n_2) := (k_{10}, k_{21}, k_{02})$.

The search structure is essentially a map to store and retrieve (key,value) pairs. In this case the key is a directed edge, represented as a pair of integers that can be lexicographically ordered. The value is the face that has entered the edge. Storage and lookup can be done in time $\log n$, when for instance the map is implemented over a balanced tree [CLR90]. Both passes can be performed in $O(n \log n)$, which also gives an $O(n \log n)$ algorithm in total.

The same task can also be performed in linear time, making use of the fact that the average vertex valence is six: A fixed size bin is created for each vertex, and the neighbours are sorted into the bins in the first pass, and retrieved in the

second. – In any case, a third pass over the triangles, where each triangle enters itself into the oneFace field of its three vertices, eventually concludes the detection of neighbourhood information.

This algorithm assumes that the mesh is a valid manifold mesh, possibly with boundaries. It can also be used, however, to detect nonmanifold configurations, i.e., some of the topological pitfalls from table 2.28 in chapter 2:

- In pass 1, it is possible to check whether the key already exists before entering an edge $(a, b) \rightarrow k$. If the same edge $(a, b) \rightarrow k'$ has been entered before from a different face, this means that there is either an edge with multiple sheet (a complex edge), or the mesh is non-orientable.
- In pass 2:, If there is no neighbour for a given edge, then the respective triangle has a boundary edge.

Complex edges, boundaries, and non-orientability can be detected easily. But to assert that the mesh is manifold a third criterion has to be checked, the manifold vertex property from the mesh consistency theorem 2.24. This is not so simple, however, since it involves to arrange the faces incident to each of the vertices in a single cycle. Equivalently, if triangle f points to vertex v , and v references face f_v , then f and f_v must be in the same cycle around v .

C++ templates and the STL. The implementation of the above algorithm is a function collectFaceNeighbours, shown in Fig. 4.3, that builds up on the shared vertex data structure from Fig. 4.2. The implementation is greatly facilitated by the use of the *standard template library*, the STL [SL95]. The STL is a very particular library, since it is based on a design document that describes only the required properties of a number of basic data structures; it specifies only access methods that use *iterators*, rather than prescribing a particular implementation. The STL makes heavy use of templates. The template facility of C++ is a method to realize *generic programming*. This means that the basic low-level data structures, such as arrays (STL template class vector), doubly-linked lists (list), and dictionaries (map), are *parameterized* in terms of the data types they carry. The data structures themselves have to be implemented only once. They are therefore also called *container data structures*: the data to be managed are regarded as anonymous containers, and the data structure itself is realized in a generic way. Through this separation it can warrant the data structure integrity on the management level, and make sure that references etc. are always consistent.

Without going into details (see [ES90] for the exact definition of template semantics) is the simplest way to understand C++ templates as a cut & paste mechanism: The source code of the linked list itself is something like a form where the data type to be stored is just referred to as some anonymous class T. When a concrete list is needed, for example a list of a particular class of vertices, the list is *instantiated*, and the T is replaced by the given type argument almost verbatim: The class list<Vertex> can be used just as if the linked list source code had Vertex instead of T everywhere in it. The fact that list<Vertex> is a template specialization can be completely hidden by assigning a different name to it, which is highly recommended: With typedef list<Vertex> VertexList, the new type VertexList can be used just like a normal list class with a standardized set of access methods.

4.1.2 Shared Vertex Mesh with Traits

The first version of the shared vertex implementation from Fig. 4.2 is not very flexible. The size of the vertex and face arrays is fixed to 100, but this could be easily remedied by dynamic memory allocation (malloc in C, or new in C++). More serious is the fact that the vertex and face classes are a combination of both connectivity and geometry information (vertex positions). Recall from chapters 2, and especially from the mesh definition 2.30, that it is quite reasonable to distinguish between the ‘abstract’ mesh, as defined by the entities and the incidence relationships, and its embedding.

Besides this theoretical justification for a separation on the data structure level, there is also a very practical one: It is very desirable to have a clear distinction between the connectivity of the mesh on the one hand and the *attributes* of vertices, edges, and faces, on the other. If this is not so, the vertex and face classes themselves have to be changed whenever a different set of attributes is needed. Different algorithms and applications may require completely different sets of attributes, while still using essentially the same mesh data structure.

Different sets of vertex and face attributes. The object in in Fig. 4.2 (b) had to be rendered in wireframe mode because the mesh does not provide surface normals so far. Normal vectors are indispensable for lighting calculations, to render objects with different materials, illuminated by a number of different light sources etc. Surface normals are usually specified per vertex for meshes approximating smooth surfaces. Vertex normals from the tessellation of a sphere, for example, all point radially outside, they point from the sphere center into the direction where the vertex is. Using the normal the graphics hardware calculates a color value at each vertex, according to a chosen shading model. Shading requires only the orientation of the surface, not the surface itself, to determine the color of a piece of surface around the vertex that is infinitely small. The color of points in the interior of the triangles is usually derived from the colors of the vertices, e.g., by linear interpolation (‘Gouraud shading’). Surface normals are only one example of possible entity attributes. Further examples include face normal vectors, face or vertex colors, and texture coordinates for vertices.

```

template<class Trait>
struct VertexSVN : public Trait::V
{
    typedef FaceSVN<Trait>    Face;
    typedef typename Trait::V V;

    VertexSVN() {}
    VertexSVN(const V& data) : Trait::V(data) {}
    ~VertexSVN() {}

    FaceSVN<Trait>* oneFace;
};

template<class Trait>
struct FaceSVN : public Trait::F
{
    typedef FaceSVN <Trait>    Face;
    typedef VertexSVN<Trait>   Vertex;
    typedef typename Trait::F F;

    FaceSVN() {}
    FaceSVN(const F& item) : Trait::F(item) {}
    ~FaceSVN() {}

    Vertex* vertex [3];
    Face*  neighbour[3];
};

template<class Trait>
struct MeshSVN : public Trait
{
    typedef VertexSVN<Trait>   Vertex;
    typedef FaceSVN<Trait>    Face;

    MeshSVN() { vOld = vertices.begin();
               fOld = faces .begin(); }
    ~MeshSVN() {}
    bool checkForRelocation();

    vector<Vertex> vertices; Vertex* vOld;
    vector<Face>   faces;   Face* fOld;
};

struct TraitSVN_Vtpn_Fm
{
    typedef TraitSVN_Vtpn_Fm Trait;
    typedef MeshSVN <Trait> Mesh;
    typedef FaceSVN <Trait> Face;
    typedef VertexSVN<Trait> Vertex;

    struct V
    {
        V() {}
        V(const Vec3f& p) : position(p) {}
        virtual ~V() {}

        Vec2f texturecoord;
        Vec3f position;
        Vec3f normal;
    };

    struct F
    {
        F() {}
        F(const F& f) {}
        virtual ~F() {}

        Vec3f faceMidpoint () const;
        Vec3f normal        () const;
        Box3f bbox          () const;
    };

    int materialID;
};

```

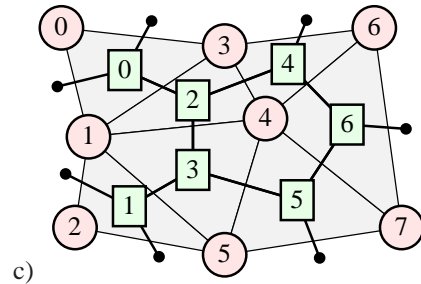


Figure 4.4: Shared vertex implementation using traits. Basically the same data structure as in Fig. 4.2, but split in two parts: generic mesh as container data structure (left), and trait class for vertex and face attributes (right).

```

template<class Trait> bool TriMesh<Trait>::checkForRelocation()
{
    Vertex* vNew = vertices.begin();
    Face* fNew = faces .begin();

    if (vOld==vNew && fOld==fNew) { return false; }

    for (vector<Face>::iterator f=faces.begin(); f!=faces.end(); ++f) {
        for (int k=0; k<3; k++) {
            if (f->vertex [k]!=NULL) { f->vertex [k] = vNew + (f->vertex [k] - vOld); }
            if (f->neighbour[k]!=NULL) { f->neighbour[k] = fNew + (f->neighbour[k] - fOld); }
        }
    }
    for (vector<Vertex>::iterator v=vertices.begin(); v!=vertices.end(); ++v) {
        if (v->oneFace!=NULL) { v->oneFace = fNew + (v->oneFace - fOld); }
    }
    vOld = vNew;
    fOld = fNew;
    return true;
}

```

Figure 4.5: Relocation repair function. It must be called whenever dynamically adding/removing entities, since STL vectors may unexpectedly relocate to a different memory location, invalidating all pointers to vertices/faces.

Generic programming using traits. A separation of the mesh data structure from the data the mesh entities have to carry can be achieved in a similar way as in the STL by using templates. This turns a mesh class into a container data structure. The mesh itself is essentially just a graph, and its task is to keep the incidence relationships consistent. The containers are the entity attributes to be stored in the mesh. They can be seen as a more general form of an embedding, into an abstract ‘data space’ instead of \mathbb{R}^3 .

Unlike linked lists, however, meshes have three different types of entities, so a mesh template has to be parameterized in terms of three different types of containers. For a few serious technical reasons (scoping, debugging, etc.), it is more than desirable to have only a single template parameter – usually the attribute classes are not independent from each other anyways. This can be accomplished by the ‘local type’ facility offered by C++: Not only can a class have local type declarations, equally possible is to declare new ‘local’ classes inside of its scope. Such a class with sub-types, that is used as template parameter, is called a *trait*.

Definition 4.2 (Trait)

A trait is a C++ class that is (a) used to instantiate a template, and that (b) contains local type names required by the template. The purpose of the local type names is to further qualify the properties of the instantiated template. The local types may come from typedef statements or from class declarations within the scope of the trait.

The shared vertex data structure as a template. The code to the left in Fig. 4.4 directly corresponds to the previous implementation of shared vertex meshes, but now the implementation is generic. Vertices and faces are implemented as template classes, derived from the respective attribute classes. The trait class `TraitSVN_Vtpn_Fm` is shown to the right. It codes in its name the properties of vertices and faces: a vertex contains position, normal vector, and 2D texture coordinates, and a face has a material identifier.

The names of the attribute classes, `Trait::V` and `Trait::F`, are prescribed by the mesh template. Note that the mesh template in no way assumes that, e.g., a vertex must contain a 3D position. The mesh template only standardizes the access to the mesh entities and the connectivity information.

- The mesh template provides five types `Mesh`, `Vertex`, `Face`, `V`, and `F`, that always refer to the mesh itself, a mesh vertex, a mesh face, and the vertex and face attributes.
- The connectivity information is always stored in `Vertex::oneFace`, `Face::vertex[]`, and `Face::neighbour[]`.
- The vertex and face classes are derived from the attributes, so both types have compatible pointers. As the attribute classes have *virtual* functions (the destructor), an attribute pointer can be dynamically casted to an entity pointer. If `faceattribute` is a pointer to a face attribute (of type `Face::F*`), then the respective `Face*` pointer is obtained like this:

```
Face* face = dynamic_cast<Face*>( faceattribute );
```

- Note that the attribute classes can not only attach additional data members to a mesh entity, but they can also provide new member functions. Attribute member functions even have access to the connectivity information: A dynamic cast can be also applied to the `this` pointer of a `Trait::F` object.

The ability to turn an attribute pointer into an entity pointer has great significance for the efficiency of the template approach. This is a difference between meshes and, say, linked lists, even when both are container data structures: While it is rarely necessary that a list item needs access to the previous and next elements in the list, this is very well the case with mesh items. Otherwise decent functionality like `faceMidpoint` or `boundingBox` functions could not be realized as attribute member functions. But this is exactly where they belong, since only the trait knows where to find, e.g., the vertex position.

But note that this feature is only optional: The template classes `TriVertex` and `TriFace` do not have virtual functions. If the respective attributes `Trait::V` and `Trait::F` do not introduce virtual functions, one pointer (4 bytes on 32 bit architectures) is saved per vertex and face. But without any virtual functions, no dynamic cast to a derived class is possible [ES90].

References: Pointers versus indices, and the relocation problem. The new version of the shared triangle mesh uses pointers instead of indices as reference to vertices and faces. But it also uses STL vectors as dynamic arrays to store vertices and faces. It is important to keep in mind that this combination can cause serious trouble.

The STL vector container is specified as a dynamic list that nevertheless permits random access in constant time to each element [SL95]. Most vector implementations realize this as an array, located at the beginning of a coherent chunk of memory. The chunk is a bit larger than the array, so that elements can be appended, e.g., using `push_back`. When the space is used up, the whole chunk is copied to a newly allocated chunk of twice the old size, and the old chunk is released. This operation is called *relocation*.

In theory, pointers and indices are equivalent, since one can always convert from one to the other – but a conversion always introduces an overhead. So in practice, this is a design choice with subtle, but important consequences: Pointers are absolute memory positions, while indices are only relative.

- **Contra indices:** An index alone is useless and confusing: To which array does it refer? The beginning of the array, i.e., the memory address of the 0-th element, must be known in order to access the target element.
- **Pro pointer:** Pointers can be used to get directly hold of one specific element. A face member function can for instance call a member function of one of its vertices, without access to the mesh.
- **Pro pointers:** Type safety. It is not possible to take a face pointer for a vertex pointer without an explicit cast. But indices can stand for anything.
- **Contra pointer:** When an array moves to a different location, the original objects usually remain intact for a while, and errors can be deferred to the moment when the memory is reused and a defunct object is accessed.
- **Pro indices:** Array positions are very robust against array relocation. Copying or moving an array to a different memory location leaves the relative positions within the array intact.
- **Pro indices:** Index arithmetic. There are more than 4 billion combinations of 32 bits. It is often possible to sacrifice bits to store additional data, for instance the sign bit, so that negative numbers have a special meaning.
- **Contra index arithmetic:** It affects scalability. Today meshes exist with billions of vertices. An ‘abuse’ of single bits is not very robust, and must be well documented. Much cleaner is to use an explicit status bit-field, into which special states can be coded.

In summary, one can keep in mind that pointers are more convenient, especially from an object-based point of view, but indices are more robust against copying and memory relocation. In practice, also the purpose of the data should be considered: Some APIs are index based, such as OpenGL’s indexed geometry, while others are pointer based, such as many OS services and standard library functions like the string functions. The indexed triangles of shared vertex meshes can be directly passed on to OpenGL (Fig. 4.2, right), this is not possible with the pointer based version.

A solution to the relocation problem for meshes. Concerning the relocation problem, it is important to take precautions: Whenever operations change the size of a vector (increase or decrease it), it is necessary to check whether the array was relocated. This is done by storing the beginning of the vector in `beginOld`. If the beginning has changed after an operation, all pointers `p` referring to objects in the vector need to be updated as `p=beginNew+(p-beginOld)`. Note that taking the difference `beginNew-beginOld` as a memory offset would be faster¹, but this is illegal, since pointer differences are only allowed for pointers into the same array [ES90]. A relocation may look like excessive overhead, and like an argument against using vectors. But note that relocations happen only to double the vector size, so a vector of size n has had to undergo only $\log n$ relocations.

For meshes with pointer references a relocation of the vertex or face arrays destroys the mesh connectivity. The relocation check must consequently be performed each time when a vertex or face is added. The implementation of the `TriMesh<Trait>::checkForRelocation` routine of the mesh from Fig. 4.4 is shown in Fig. 4.5. After every relocation, it loops through both arrays, so its cost is $O(|F| + |V|)$. This happens whenever either of the two vectors is relocated, so the total cost is less than $O((|F| + |V|) \cdot (\log |F| + \log |V|))$, which equals of course $O(n \log n)$ for $n \in \{|V|, |E|, |F|\}$. – Consequently, such a routine should always be provided when using dynamic arrays together with pointer references, to solve the relocation problem.

4.1.3 Halfedges and Mesh Iterators

Edges are not explicitly represented in the shared vertex data structure. This is an important design decision. It has consequences for mesh manipulation, as well as for the possibility to navigate over a mesh.

The position in a mesh: Halfedges. A mesh is composed of entity sets V , E , and F , and the relations between the entities in these sets. So what is a ‘position’ in a mesh? A face is a unique entity, but it may have many vertices. A vertex is unique, but it may have many edges. An edge is unique as well, but it has two endpoints. So only a combination of entities unambiguously denotes a position. Generally a pair of incident entities is enough to derive the third: A (face,vertex) pair for example denotes a unique edge, by convention for instance the one in CCW direction in the face. In some cases this is still not unique: A vertex can be several times part of the same face, as in the torus example (or the double torus, Fig. 2.7).

Both vertices and faces can be incident to a variable number of entities, but – at least in the manifold case – edges are different, since every edge is incident to exactly two faces and two vertices. So edges are a good instrument for navigating over a mesh: Either of the two ‘sides’ of an edge unambiguously identifies one of two possible (face,vertex) pairs. Since there are two such pairs, the edge has two sides, each of them referred to as one *halfedge*. Halfedges use to be depicted as half-arrows (Fig. 4.6 a).

¹The memory alignment problem can be solved by casting: `char* d=(char*)beginNew)-(char*)beginOld); p=(Object*)((char*)p+d);` This usually works, even on segmented memory architectures, in contrast to what is said in [ES90]

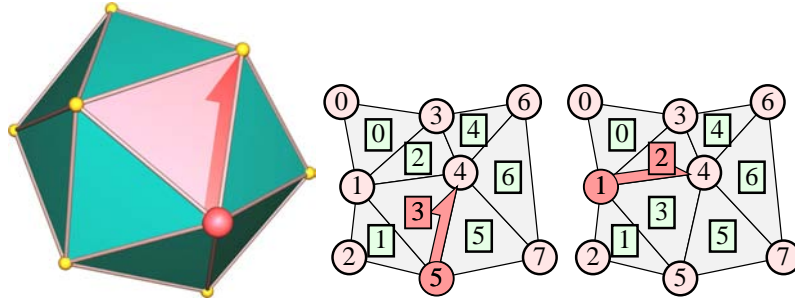


Figure 4.6: Iterating over a triangle mesh. (a) Manifold edges have two distinct sides, and each side can be associated with one face and one vertex, for example the vertex the halfedge emanates from (source vertex, red). The direction of a halfedge is induced by the CCW face orientation, so halfedges are shown as ‘half-arrows’. (b-e): Crawling over the mesh with halfedge navigation functions, like *faceCCW* (b,c), *mate* (c,d), and *vertexCCW* (d,e).

In the shared vertex data structure a halfedge is referred to by a combination (f, i) of a face f and a neighbour index i , $0 \leq i \leq 2$. As shown in Fig. 4.2 (a), index i refers to a vertex and the next neighbour face in CCW direction, incident to vertices $f.v[i]$ and $f.v[i+1]$. This is a matter of convention, though: sometimes halfedges are defined to refer to the target rather than the source vertex.

Mesh navigation: Halfedges as mesh iterators and for path expressions. An object that refers to a position in a data structure, such as a specific element in a list, is commonly referred to as an *iterator*. Halfedges can be understood as mesh iterators: Just like a list iterator l permits to traverse a list by $l' = l.pred$ or $l' = l.next$, a mesh can be traversed using halfedges. Unlike lists, meshes are locally two-dimensional graphs that, in a way, also incorporate their dual graphs: Recall from the duality definition (Def. 2.28) that edge cycles of vertices and faces are mutually topological duals of each other. Consequently, there are five canonical navigation operations for a halfedge, or mesh iterator, h :

- Traversing the edge cycle of a vertex in either direction with $h.vertexCCW$ and $h.vertexCW$
- Traversing the edge cycle of a face in either direction with $h.faceCCW$ and $h.faceCW$
- Moving to the halfedge on the opposite side with $h.mate$.

Of course, $h.vertexCCW.vertexCW = h$ and $h.faceCCW.faceCW = h$ and vice versa, and $h = h.mate.mate$. As an example for mesh navigation consider the triangles in Fig. 4.4 (c). In this case it is possible to use a (face,vertex) pair instead of a (face,integer) pair to denote a halfedge. So consider triangle f_3 with vertices (v_1, v_5, v_4) , and let $h = (f_3, v_5)$. Then $h' := h.faceCCW = (f_3, v_4)$, $h'' := h'.mate = (f_2, v_1)$, and $h''' := h''.vertexCCW = (f_0, v_1)$. The resulting path over the mesh is illustrated in Fig. 4.6 (b-e). Slightly abusing C++ style *path expressions*, this can also be concisely expressed as:

$$(f_3, v_5).faceCCW.mate.vertexCCW = (f_0, v_1)$$

One of the requirements for mesh data structures is that these operations are performed very fast, ideally in constant time. With the shared vertex data structure, *faceCW* and *faceCCW* are just a matter of computing $(i-1) \bmod 3$ and $(i+1) \bmod 3$, respectively. When the *mate* operation is available, *vertexCW* and *vertexCCW* can be realized as

- $h.vertexCW = h.mate.faceCCW$
- $h.vertexCCW = h.mate.faceCW$

Note the reversal of orientations due to the duality of vertices and faces. – So the *mate* operation is indeed quite useful.

Problems of the shared vertex data structure. One consequence of the design may be not so obvious: There may be problems to determine the opposite halfedge, the *mate* (g, j) of a given halfedge (f, i) , in case of degenerate triangles.

Definition 4.3 (Degenerate triangle)

A triangle is topologically degenerate if it references two times the same vertex, or two times the same face, or even both. It is geometrically degenerate if it collapses to a line or a point because different vertices share the same position. Consequently, a double-sided triangle is degenerate, as well as a pair of triangles forming an ‘ear’, i.e., triangles that are mutual neighbours on two of three edges.

Let (f, i) be an edge of a topologically degenerate triangle for which the mate (g, j) is to be determined. The face g on the opposite side is readily found as $g = f.n[i]$. To find j , the edge must be found that f shares with g . It is not sufficient to determine j as the neighbour of g pointing to f , since this is not unique when f references two times the same face. To be

```

template<class Trait>
struct MeshDE : public Trait
{
    typedef MeshDE<Trait> Mesh;

    struct Vertex : public Trait::V
    {
        int oneEdge;
    };

    struct Edge : public Trait::E
    {
        Edge* Edge::mate(Mesh& mesh) {
            return &mesh.edges[mate];
        }
        int vertex, mate;
    };

    typedef Trait::F Face;

    vector<Vertex> vertices;
    vector<Edge> edges;
    vector<Face> faces;
};
    
```

face	face 0			face 1			face 2			face 3		
edge	0	1	2	3	4	5	6	7	8	9	10	11
vertex	0	1	3	1	2	5	1	4	3	1	5	4
mate	-1	8	-1	-1	-1	9	11	12	1	5	15	6

face	face 4			face 5			face 6		
edge	12	13	14	15	16	17	18	19	20
vertex	3	4	6	4	5	7	4	7	6
mate	7	20	-1	10	-1	18	17	-1	13

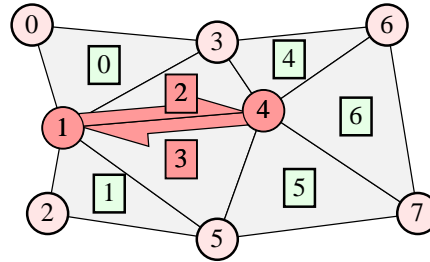


Figure 4.7: Directed Edge triangle mesh. Left: Half-edges are explicitly represented by the type Edge. It is derived from Trait::E, so halfedges can also carry information. Right: A mesh with 7 faces corresponds to an edge array of length 21. Every 3 halfedges implicitly belong to one face, so that the halfedge can refer to its mate directly: Edge 6 is incident to (v_1, f_2) , its mate edge 11 to (v_4, f_3) .

safe, it must additionally be checked whether $g.v[j] = f.v[i + 1]$ and $g.v[j + 1] = f.v[i]$. In this case j is unique provided that the triangles are not ‘extremely’ degenerate, i.e., not all of the vertices coincide.

Degeneracies like ears and double-sided triangles, or coincident vertex references, are special cases that need to be checked in order to avoid errors. The problem with special cases is not only that they are sometimes hard to detect, but also that to check for them binds computing resources: In practically all cases the triangles are not degenerate and there is only a single j so that $g.n[j] = f -$ but to be sure, one has to check *always*. Another possibility is to design algorithms in a way that the special cases can (provably) not occur.

Both alternatives are just as error prone as they are tedious. In the long run, it is much better to design the data structure more carefully, namely in a way that special cases are no longer special cases. This may require more effort, but it pays off as the resulting data structures are more sustainable. Taking the STL as model, the specification of a mesh data structure should rather be formulated in terms of the operations on the mesh and their costs, than on the objects involved. The set of mesh operations should of course be closed and sufficient (c.f. theorem 2.29).

4.1.4 Beyond Shared Vertices: Campagna’s Directed Edges and Hoppe’s Wedges

The problem with the shared vertex data structure is that neighbourhood information is only stored on a per face basis: The neighbour face is directly accessible, but to determine which of the neighbour’s edges is the mate of the given edge requires some testing. To make these tests robust against degeneracies causes even more overhead. This is hardly tolerable, since the *mate* operation is used very frequently.

Directed edges. This issue can be solved if the halfedges, which have proven quite useful in the last section, are explicitly represented in the data structure. This appears like a large space overhead, because there are so many of them – six times as many as vertices (Theorem 4.1). But note that a shared vertex mesh also stores six references per face, to three vertices and to three neighbour faces, which is nothing but two references per halfedge.

With a simple trick, the space requirement remains the same, but halfedges can be referenced directly: Just store the mate halfedge (f_i, j) with $j \in \{0, 1, 2\}$ as the integer number $k = 3 \cdot i + j$ in the neighbour field, instead of just storing i . This combined number k is called a *halfedge index*. It can be directly used as an array index if all halfedges are contained within one coherent array: The face is split up into three (vertex, halfedge index) pairs, the *directed edges*. The resulting *directed edge data structure* was introduced by Swen Campagna [Cam98, CKS98]. It appears that it has been re-invented a number of times, in any case it is probably *the* standard triangle mesh representation today.

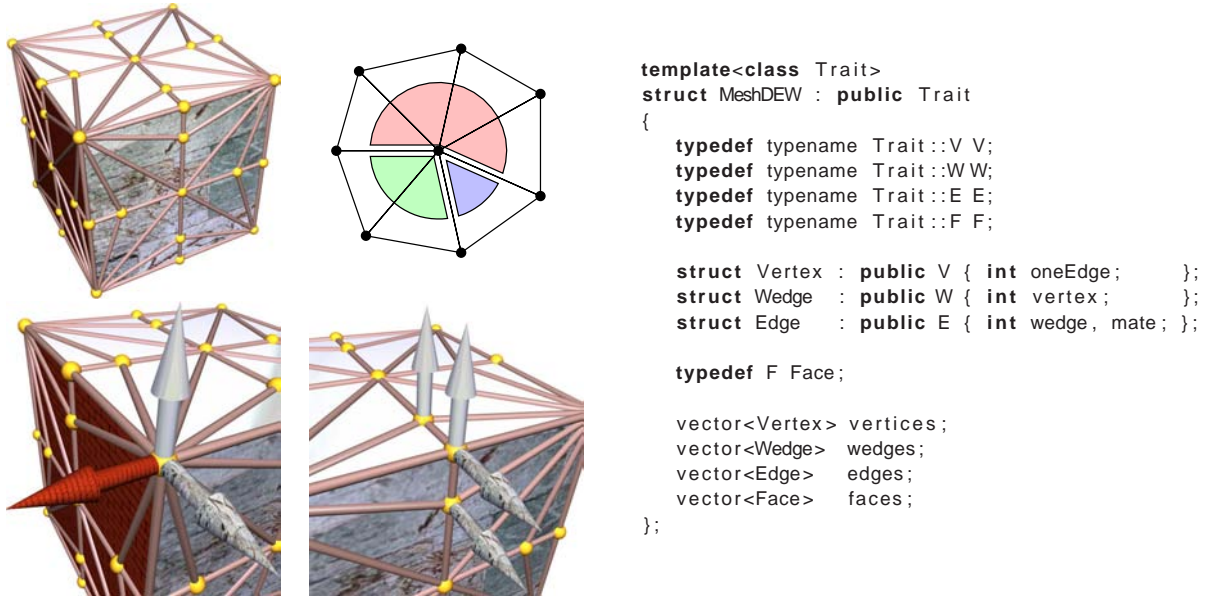


Figure 4.8: The wedge problem. Left side: (a) Some vertices lie in the interior of a surface, others are corners, or lie along creases between surface patches. (b) Wedges are consecutive corners around a vertex. (c) A corner vertex adjacent to 12 different triangles, and with 3 different attribute sets. (d) Interior vertices have only a single face normal, vertices on a crease have two. Right side: The directed edge data structure where halfedges do not reference vertices, but wedges, which then reference vertices.

Campagna argues that it is worthwhile to trade the space for storing a mate index against a few computations to obtain it.

- Halfedges $3i$, $3i+1$, and $3i+2$ all belong to triangle i ,
- halfedge k directly stores a reference to its mate k' , and
- the mate k' belongs to triangle $k'/3$ (integer division rounds down).
- $faceCCW$ is realized by $(k \% 3 == 2) ? k - 2 : k + 1$
- $faceCW$ is realized by $(k \% 3 == 0) ? k + 2 : k - 1$

The already familiar example mesh is shown in Fig. 4.7. Note that the type `Edge` actually denotes a halfedge. The function `Edge::mate()` shows how path expressions can be used with index references: To return an `Edge*`, the function needs access to the mesh. – In the diagram, the edge between vertices 1 and 4 is highlighted. The two respective halfedges can be found in positions 6 and 11 in the edge array (table). They mutually refer to each other as mates. From the indices alone it is clear that they belong to faces $6/3 = 2$ and $11/3 = 3$, and the vertex fields of these halfedges reveal that they refer to vertices 1 and 4. The connectivity is entirely defined by the halfedges. With this method the connectivity is also robust against degenerate triangles: There are no special cases to take care of.

Note that there are different ways to store face number i and index j together. With $4 \cdot i + j$, the lower two bits code the index, so that division and modulo can be replaced by bit shift and masking, which is faster. But then, the combined number can no longer be directly used as an array index. This is an example of coding additional information into an integer reference, which is not possible with pointers. Another application, also pointed out by Campagna, is to use the sign bit as a flag for non-manifold configurations: While a mate value of -1 denotes a border edge, a value of -2 might be used to mark complex edges. A negative value $-k$ of the `oneEdge` field can flag a complex vertex, and k is used as an index into an array with a set of halfedge indices, one for each edge cycle the complex vertex is incident to. The possibility to represent non-manifold configurations is very important, e.g., for mesh simplification using the *non-edge collapse*, which will be presented in the next section.

The wedge problem, and Hoppe's solution to it. A very particular problem has not yet been considered: A vertex may very well have more than one normal vector attached to it. Fig. 4.8 (c) and (d) illustrate the problem. Most of the vertices use to lie in the interior of a surface region, where they have only a single normal vector, and only a single vertex attribute. A group of vertices that are to approximate a smooth surface patch is also called *smoothing group*. The boundary where different patches meet may be discontinuous to form a *crease* in the surface. Consequently, the tessellation may comprise vertices with different sets of attributes, depending on which face the vertex is used with: The normal vector, but also for instance a vertex color, or texture coordinates, are conceptually per-vertex-per-face attributes.

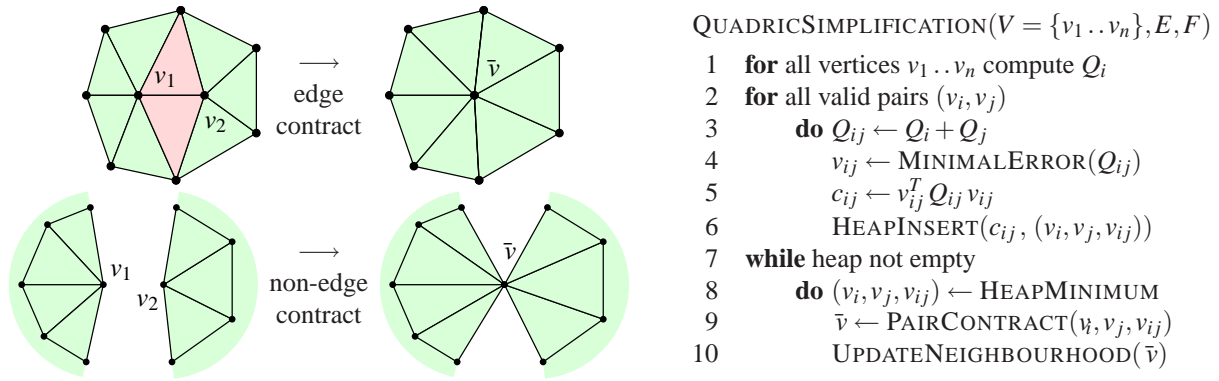


Figure 4.9: Two forms of pair contraction. Right: The edge contract merges two triangle vertices into one, thereby removing two triangles, three edges, and one vertex from the triangulation. A non-edge contract creates a complex non-manifold vertex from a pair of vertices that are close but not connected via an edge. It can merge different shells together and bridge topological holes. Left: Outline of the simplification procedure.

One solution would be to store the vertex attributes in the halfedges, rather than the vertices. But besides the large number of halfedges, the space overhead is excessive, since the average number of different attribute sets per vertex is much lower than the average vertex valence. So this raises again the question how to handle special cases without impairing the average case. The solution proposed by Hugues Hoppe in [Hop98] is to insert one layer in between vertices and faces, the *wedge*. A wedge is a part of a vertex's edge cycle, i.e., a number of consecutive edges around a vertex. Each edge cycle is composed of a number of wedges, every two consecutive wedges sharing a common edge. The image Fig. 4.8 (b) shows three wedges, made of four, two, and one corners. A *corner* is just a vertex of a face, plus the two respective edges of the face boundary.

Faces (or halfedges) do not refer directly to vertices, but to wedges, and all wedges of one cycle of course refer to the same vertex. Wedges carry the attributes of the vertex with respect to the faces of the wedge, i.e., with respect to the corners. The source code to the right in Fig. 4.8 demonstrates that the space overhead is one reference per vertex, the reference from the wedge to the vertex. This means that a mesh with v vertices and f faces requires $(5 + 2 \cdot 6)v = 17v \approx 8.5f$ references, or $68v \approx 34f$ bytes, i.e., 34 bytes per triangle. This is 2 bytes/ Δ more than with a directed edge data structure, but with the latter multiple attributes per vertex are simply not possible.

4.1.5 Automatic Mesh Simplification using Error Quadrics

The purpose of automatic mesh simplification is to remove redundant mesh entities to obtain a mesh that is smaller than the given mesh. The crucial question is: What is a good approximation? And which entities are redundant? The answer is usually given in terms of an error measure that quantifies the distance between the original mesh and the simplified version. Simplification is an iterative process, and by monitoring the error, a tolerable approximation can be found.

The idea of the approach from Garland and Heckbert is to track for each vertex the sum of the squared distances from the vertex to a set of planes from surrounding triangles [GH97]. In the beginning this distance is zero for every vertex. Then pairs of vertices are iteratively merged into one, either by contracting an edge, or by a non-edge contraction in case two vertices are very close but not connected via an edge (Fig 4.9, left). The error functions of both vertices are combined, so that the combined vertex inherits all the planes from the merged vertices. The error of the combined vertex is not zero, since it is generally not the case that all the planes from the merged vertices intersect in a single point.

The pseudocode in Fig. 4.9 gives an outline of the algorithm. Every vertex is assigned a quadric Q that is generated from the plane equations of the incident triangles. The distance from a vertex v to a plane can be expressed as a scalar product if the vertex position p is given as 4-dimensional vector $p = (x, y, z, 1)$ in homogeneous coordinates. Let $q = (a, b, c, d)$ represent the plane defined by the equation $ax + by + cz + d = 0$, with $a^2 + b^2 + c^2 = 1$ so that (a, b, c) is the normalized plane normal vector.

The signed distance from p to q is $q^T p = (a, b, c, d)^T \cdot (x, y, z, 1)$. When this distance is squared, it can be expressed as a quadratic form Q that is the dyadic product of the plane equation with itself:

$$(q^T p)^2 = (p^T q)(q^T p) = p^T (q q^T) p =: p^T Q p = (x, y, z, 1) \begin{pmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

When v has valence k , it is incident to k faces with plane equations q_1, \dots, q_k , and the sum of the squared distances is

$$\sum_{q_1 \dots q_k} p^T (q_i q_i^T) p = p^T \left(\sum_{q_1 \dots q_k} q_i q_i^T \right) p = p^T \left(\sum_{Q_1 \dots Q_k} Q_i \right) p.$$

Instead of explicitly storing all the plane equations associated with a vertex they are simply summed into a single quadric Q . A quadric is a symmetric 4×4 matrix, and it is positively definite, i.e., $p^T Q p \geq 0$ for all p . It can be stored using only the 10 floats for the upper half, and the same is also true for the sum of the quadrics.

The main idea of the simplification is to assign a cost to every potential contraction. So the algorithm proceeds by assigning cost values c_{ij} to all edges in the mesh. Vertices that are not connected by an edge are also evaluated, but only if they are very close to each other, i.e., their distance is smaller than a preset threshold. Together they form the set of *valid pairs*. The error that is introduced when a pair (v_i, v_j) of vertices is collapsed into a single vertex \bar{v} can be measured as the capability of this point to satisfy simultaneously all the plane equations that v_i and v_j have to satisfy. One way to achieve this is by simply adding the quadrics from the vertices of a valid pair to obtain $Q_{ij} = Q_i + Q_j$. So when a pair (v_i, v_j) is contracted into a single vertex \bar{v} at position \bar{p} , the error introduced is $\bar{p}^T (Q_{ij}) \bar{p}$. This is defined as the cost of this contraction. Note that the two triangles incident to the edge are counted twice when adding the quadrics. Garland and Heckbert argue that this is tolerable.

Of course, the position \bar{p} has to be chosen in an optimal way, so as to minimize the cost. But a quadric, or quadratic form, is just a function $c: \mathbb{R}^3 \rightarrow \mathbb{R}$ that is quadratic in x , y , and z :

$$c(p) = p^T Q p = a^2 x^2 + b^2 y^2 + c^2 z^2 + 2abxy + 2bcyz + 2aczx + 2adx + 2bdy + 2cdz + d^2$$

This means that the constant-error iso-surface $E(\epsilon) = \{p \in \mathbb{R}^3 \mid c(p) = \epsilon\}$ is an ellipsoid. Every (non-degenerate) quadratic function has a unique minimum, so there is only a single point $(x, y, z, 1)$ for which the gradient is zero:

$$\begin{pmatrix} \partial c / \partial x \\ \partial c / \partial y \\ \partial c / \partial z \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = 2 \cdot \begin{pmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \stackrel{!}{=} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Assuming that the 3×3 submatrix from the first three columns is invertible, the minimum point \bar{p} can be readily computed by solving a linear system. If it is not invertible, for instance if the quadric is made of summing up identical planes, \bar{p} is chosen in the middle between v_i and v_j . The hypothetical vertex is returned by the procedure `MINIMALERROR`.

All valid pairs are sorted into a heap, according to their cost. A heap permits to quickly identify the minimum cost contraction. When this contraction is performed, Q_{ij} becomes the quadric of \bar{v} (in the procedure `PAIRCONTRACT`). All edges that before were incident to v_i and v_j then become incident to \bar{v} . Especially after a non-edge contraction it is important to clean up and remove degenerate triangles, i.e., triangles that are incident to the same vertex or neighbour face more than once. After cleanup, new incidences have been established, especially to the new vertex \bar{v} . This means that new valid pairs are formed, others have become obsolete, and the heap must be updated accordingly, as part of the procedure `UPDATENEIGHBOURHOOD` in the last line of the algorithm in Fig. 4.9).

The possibility to glue different shells together is very important if a seemingly connected object is composed of a multitude of different shells: A tree for instance might very well be modeled by replicating the leaves, so that many identical copies, or just references to a ‘master leaf’, are floating in space, which are not attached to the trees. Garland and Heckbert give the example of a set of closely spaced but unconnected cubes, which can be merged into a single big box when non-edge contractions are possible.

Both operations, edge and non-edge contractions, can change the genus of a surface. In particular, topological holes can be closed, which is important, e.g., when processing medical data sets. But as a subtlety note that also an edge contraction can lead to degeneracies, for instance when removing a topological hole. Right before removal, the hole consists of only two edges e_A and e_B , both being incident to the same vertices v_i, v_j ; but the edges are each incident to different faces. When edge e_A is contracted, v_i and v_j collapse into \bar{v} , and e_B becomes a loop. This makes both triangles on e_B degenerate, and they must be removed.

4.1.6 Progressive Triangle Meshes

Redundant mesh entities can be removed by methods for automatic simplification, such as the technique from Garland and Heckbert in the previous section. But this redundancy is very often only relative, and not absolute: When an object is inspected at a close distance, one would like to see the object at the greatest available resolution.

In 1996, Hugues Hoppe has presented an idea to make the simplification reversible [Hop96]. Entities are removed step by step, applying simple local operations such as edge and non-edge contractions. Focusing on one of them, the edge contraction, he noticed that it is invertible. This pair of mutually inverse operators was called *edge collapse* and *vertex split*, or *ecol* and *vsplit* for short, shown in Fig. 4.10.

In order to create a progressive mesh, or short *PM*, a simplification algorithm is made to continue its work until no more detail can be removed, e.g., until just a tetrahedron remains from a genus 0 object. This mesh is called the *basemesh*, and all operations applied during simplification, leading from the original mesh to the basemesh, are logged.

Definition 4.4 (Progressive Mesh)

A progressive mesh consists of a base mesh $M^0 = (V^0, E^0, F^0)$ and a sequence (r_0, \dots, r_k) of split records, the split sequence. Let $n = |V^0|$ and $m = |F^0|$. Each split record r_i permits to insert one vertex v_{n+i} , two triangles f_{m+2i}, f_{m+2i+1} , and three edges into the mesh using a vertex split, and to also remove them again using an edge collapse.

When a triangle mesh is converted to a progressive mesh, any desired resolution between the coarse base mesh and the highest resolution can be quickly adjusted. Vertices and faces can be progressively added or removed by executing the appropriate split records, i.e. by traversing the split sequence in either direction. This means that the object is converted into a *multi-resolution mesh*. One central observation is that progressive meshes take a fundamentally different view on surfaces, which is in fact a *paradigm change*: A mesh is understood as the result of applying a sequence of operations – rather than some static tables or lists of mesh entities.

Applications of progressive meshes. The basic idea has proven to be very fruitful, and a number of extensions and variations build upon progressive meshes. They have also grown into a semi-standard: Hugues Hoppe works for Microsoft research, and Microsoft's DirectX contains progressive meshes since version 5.0, so that they can be used by application programmers to provide 3D applications with level of detail control.

- **Continuous level of detail.**

A progressive mesh permits to adaptively adjust the needed level of detail in the mesh. This is extremely important for the ability to process scanned datasets, which may contain hundreds of millions, or even billions of triangles [LPC*00, IG03]. But very often only a coarsified 3D object is not sufficient, e.g., with medical application the availability of the highest resolution is mandatory, for legal reasons. The enormous amount of data from scanning is shown in Fig. 4.11.

- **Geomorphs.**

The insertion and removal of vertices and faces can be animated, which turns a progressive mesh into a 'geomorph' object. This avoids the 'popping artifacts' from suddenly appearing triangles, which disrupt the impression of a continuous, static surface. This is achieved by gradually moving a vertex to its target position before changing the topology. So the discontinuous change in Fig. 4.10 (top row) is replaced by the animation in the lower row, in either direction.

- **Progressive transmission.**

Some image formats, such as JPEG 2000 [jpe00], permit to inspect a refining preview of an image while this image is being transported over a low-bandwidth connection like a modem. The idea of progressive encoding and transmission is that supplemental data refine, but do not replace, the data already received. The same is possible with progressive meshes, replacing the wavelet decomposition by a hierarchy of unfolding vertices. Precautions must be taken against data loss: When a split record is missed, one pair of vertices cannot be created, but also none of their descendants.

- **View-dependent refinement.**

A vertex a can already be split if it exists in the mesh, and it is incident to the two vertices c and d from Fig. 4.10, to create triangles abc and adb with them (or their 'closest living ancestors', see [Hop97]). This implies a dependency relation between the vertices, which can be exploited to coarsen and refine the mesh on demand. Splits and collapses can be executed 'out of order', not only in the strictly linear order prescribed by the split sequence. This permits for *view-dependent* mesh refinement, which acts as a level-of-detail mechanism, to increase the rendering performance.

- **Lossless compression.**

Before progressive meshes, multiple *discrete* levels of detail were realized by storing several versions of the same object at different resolutions, thereby greatly increasing the size of the dataset. Progressive meshes not only provide a *continuous* level of detail, but surprisingly they can also be encoded in a way that the file size is even smaller than that of a standard indexed face set. Concerning the geometry, for instance, the 3D positions in a split record use to be in close distance, which can be exploited by delta encoding.

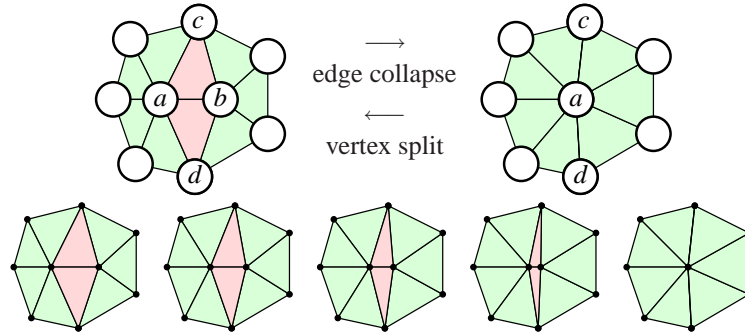


Figure 4.10: Edge collapse & vertex split, and geomorph. Top: Each split record from progressive meshes contains data to perform both edge collapse and vertex split, to coarsen and refine the model resolution. Bottom: By animating the edge collapse, popping artifacts are avoided. This is called a geometry morph, or short geomorph.

An implementation. To evaluate the approach a mesh simplifier and an implementation of progressive meshes were realized. They use a shared vertex mesh with neighbourhood pointers, parameterized by a trait, similar to the MeshSVN class from Fig. 4.2. The simplifier is based on the method from Garland and Heckbert, but it is more restricted: It uses only edge contractions (i.e., edge collapse), no non-edge contractions, and processes only closed manifold meshes. This is a serious limitation for its applicability to real-world problems. Especially meshes obtained from laser range scanning may be highly non-manifold, i.e., exhibit complications such as those listed in Fig. 2.28.

As explained earlier in sec. 4.1.3, for the shared vertex meshes to work either degenerate triangles must always be avoided, or the *mate* method to find the opposite halfedge must perform two additional checks. The choice was to do the first, as the implementation is restricted to the manifold case anyways. The restriction to the manifold case had also other consequences. It was found that two checks are important in order to avoid degenerate triangles in both the geometric and the topological sense. If either of these tests fails, the edge collapse is illegal and may not be performed:

- **Flipover test:** The face normals of the triangles incident to both vertices that are to be collapsed must be compared before and after the collapse: The collapse operation may not flip a triangle, e.g., the face normals may not deviate by more than 90 degrees before and after the collapse.
- **Triple connect test:** Suppose an edge a, b as in 4.10 is to be collapsed. The neighbourhoods of a and b share two vertices c and d . In case they share also a third vertex, a collapse results in a double edge. This can lead to a pair of degenerate triangles forming an ‘ear’.

Despite the overhead from these tests, the speed of the simplification is quite acceptable: About 10,000 vertices per second are removed on a moderate PC². For a mesh with 53858 vertices, the reduction rate is 12903 vertices/s, for a larger mesh with 94953 vertices the rate is a bit lower, 10101 vertices/s. This is due to the cost of the heap. It contains all edges of the mesh, arranged according to the cost of a potential collapse. So overall its complexity is $O(|E| \log |E|)$, i.e., it increases more than linearly.

The result of the simplification is an .obj file containing the basemesh and the split records. With a slight extension of the .obj syntax, it was possible to encode them in the same file. A split record begins with the pm tag:

$$\text{pm } i_a j_{abc} k_{adb} a_x^{\text{old}} a_y^{\text{old}} a_z^{\text{old}} a_x^{\text{new}} a_y^{\text{new}} a_z^{\text{new}} b_x^{\text{new}} b_y^{\text{new}} b_z^{\text{new}} \text{matid}_{abc} \text{matid}_{adb}$$

The tag is followed by the index i_a of the vertex a to be split, and the indices j_{abc} and k_{adb} of the faces to be created on either side of the new edge. They specify the position in the mesh where the split is to take place. Then follows the old vertex’s position, and two positions of the new vertices. The index of the newly created vertex is determined by the position of the record in the split sequence: If the basemesh has n vertices, the first split record always creates vertex $n + 1$, etc. Finally, the record contains the materials of the new faces after the split. One pm line in the file directly corresponds to a SplitRecord structure that stores it in memory, which contains five integers and three Vec3f.

The speed of the progressive meshes is of course much higher than the simplification rate. On the same machine, inserting 577114 vertices takes just 4.29 s, an insertion rate of 134,570 vertices/s. Removal is even about twice as fast, to remove 537699 vertices takes just 2.135 s, which is a rate of 251,835 vertices per second. These results were obtained by only linearly traversing the split sequence, not using a more sophisticated out-of-order or view-dependent refinement.

²500 MHz Pentium III with 256 MB RAM running Linux, gcc version 2.96 with -O3

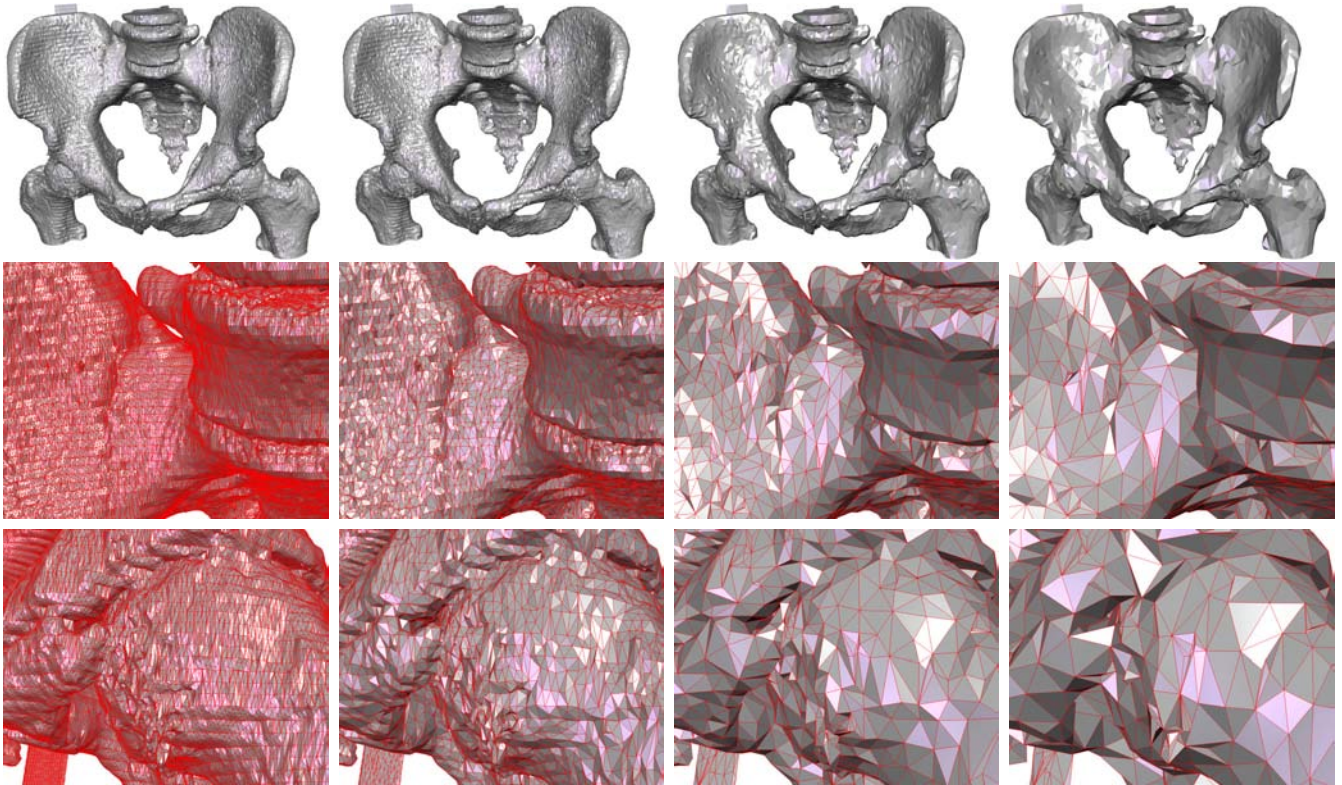


Figure 4.11: Simplification of scanned data. The original medical data set has 598,026 vertices and 1,194,668 triangles. It is reduced to 117658 (20%), 17648 (3%), and only 5882 (1%) of the vertices.



Figure 4.12: Adaptive level of detail with progressive meshes. The low resolution has only 14%, or 558 from 3992 vertices. At a low pixel resolution (a,c), the difference is hardly noticeable, despite the distortion of (b).

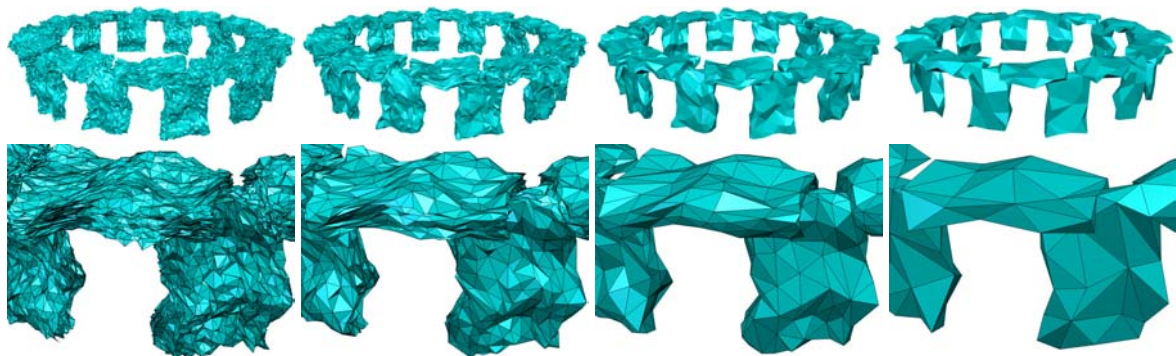


Figure 4.13: Simplification implies smoothing. The stonehenge-like object with 53855, 10770 (20%), 2692 (5%), and 538 (1%) vertices shows how low-resolution triangles are fitted to whole surface regions.

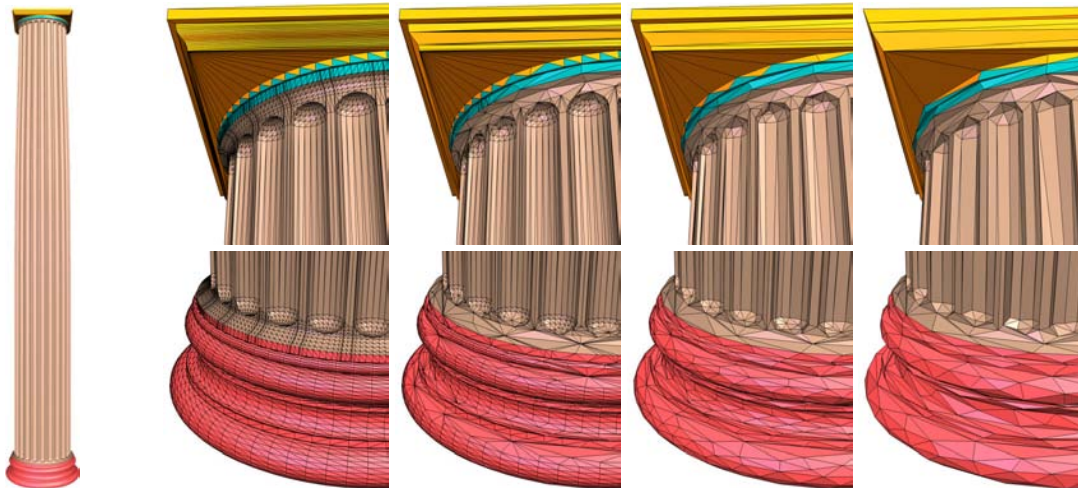


Figure 4.14: Level of detail of an ionic column. The CAD dataset has 12007 vertices, and it is reduced to 50%, 20%, and 10%, with 6003, 2401, and 1200 vertices. Note that each ridge is simplified differently.

Some results. A series of experiments was carried out to assess the efficiency of progressive meshes in conjunction with automatic simplification. The examples confirm that the method is most useful for densely sampled meshes, be they exported from CAD systems or obtained from scanning. To better judge the surface quality, flat shading was used in all images. The appearance is of course much smoother if instead vertex normals are used with Gouraud shading.

The first example is a CAD dataset, the ionic column in Fig. 4.14. It contains 12K vertices, which is good for manufacturing, but slightly oversampled for interactive visualization: A greek temple typically contains dozens of columns, and $24K\Delta$ per column is excessive. Remarkably, the column can be reduced to only 10% without much visual degradation. Only at close inspection, and with wireframes, the irregularity of the triangulation is visible. The medical dataset in Fig. 4.11 shows how mesh simplification can actually remove artifacts: The hip was scanned in slices that are clearly visible in the highest resolution. Even at 20% the slices are still present, and only at extremely high reduction rates the model becomes smooth without slices – as the bones supposedly are. This raises the fundamental question of what the ‘true’ object is. The house model in Fig. 4.12 shows the use of progressive meshes for view-dependent rendering: When the house covers only $93 \times 100 = 9300$ pixels, there is not much use for the high-resolution model with 4K vertices; a 14% version with 0.5 K vertices delivers almost an identical image, even if the low-resolution model exhibits serious distortion (b). The low-resolution images are made with 8-times accumulation buffer antialiasing. Simplification also has a smoothing effect. The Garland/Heckbert method integrates more and more plane equations into the quadrics of the mesh vertices. So after a number of steps, the vertex positions are sort of a least squares fit to a number of planes from a whole region of the original object. This fitting property then also carries over to the larger triangles, in (c) and (d).

Also with some synthetic examples simplification behaves surprisingly well. The subdivision surface in Fig. 4.15 (a) was created by extruding each quad face of a distorted L-shape. At a certain point, automatic simplification recovers exactly the original quads, and approximates the added geometry faithfully as a four-sided pyramid (b). Less surprising is that a subdivided box indeed becomes a box at some stage. The high-valence vertices in (d) are a result of the heap implementation: All edges in the plane all have identical contraction cost, but an edge from the last processed vertex is sorted to become most likely also the minimum in the next step.

Other synthetic objects reveal the limitations of automatic simplification. The window as well as the temple model in Fig. 4.16 are already low-resolution objects, with 319 and 209 vertices, respectively. Whenever vertex positions are carefully chosen so that they best represent a given object, simplification introduces noticeable distortions. Another objection is that different but symmetric parts of the same object are most likely not simplified the in same way. This applies to the faces that make up the holes in the window, as well as to the stairs in the temple model and the ionic column (Fig. 4.14). This is the reason why carefully designed discrete levels of detail are still in service today – especially when the aesthetic appearance is key, for instance in skillfully designed 3D computer games.

Issues with automatic simplification and progressive meshes. Mesh simplification and PMs are fascinating subjects of study. Especially PMs have triggered a wealth of research contributions also in related fields, from progressive encoding and transmission schemes to out-of-core simplification, progressive point clouds, and many more. Yet the experiments revealed that there are some practical, and also some fundamental issues with this technique.

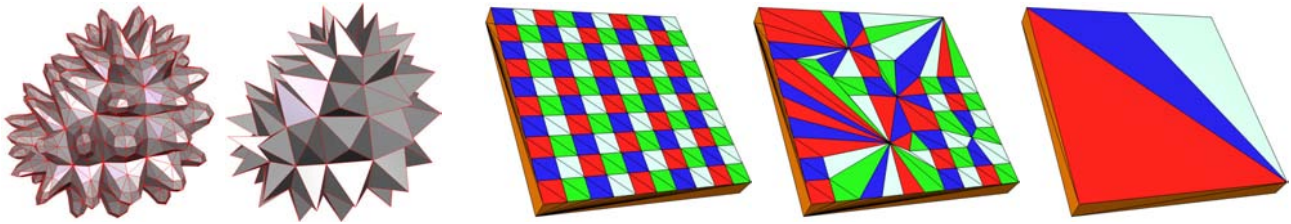


Figure 4.15: Detail preservation. At certain points, automatic simplification can reveal the ‘true’ object structure.

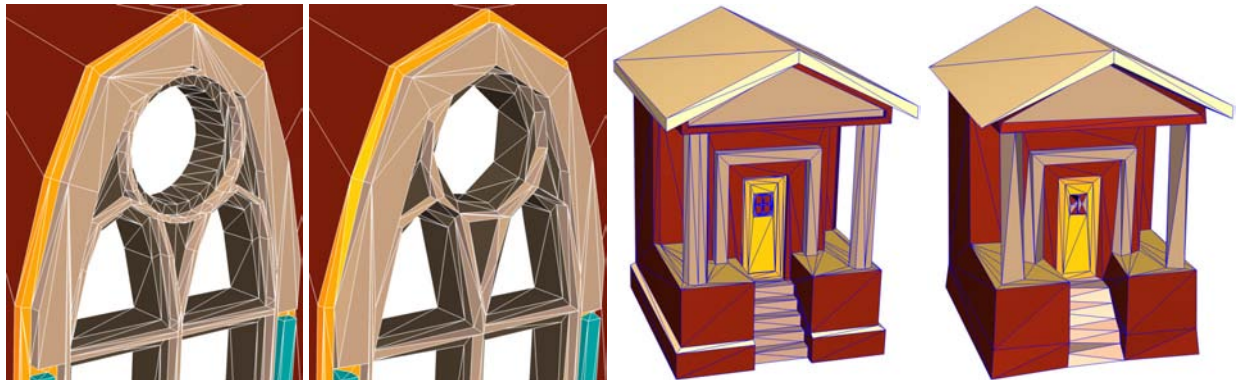


Figure 4.16: Simplification breaks symmetry.

- **Automatic simplification breaks symmetry.**

Especially when the input shape is very regular, it is not easy to simplify it without introducing irregularities. For the column model in Fig. 4.14, for instance, the fact that each of the ridges is simplified in a different way is quite annoying. The *differences* between the ridges are perceived as artifacts much before actual simplification artifacts become apparent. This is even more so with ‘light’ objects that have fewer, but more important vertices (Fig. 4.16).

- **Highest resolution is limited.**

Other than, e.g., with free-form surfaces, the highest surface resolution is limited by the resolution of the input mesh. Also at close inspection, detail cannot be synthesized beyond the input mesh resolution, for instance by a finer sampling of an underlying ‘true’ surface.

- **Per frame LOD adjustment overhead.**

View-dependent LOD tries to identify a suitable resolution on the level of individual triangles. In every frame, triangles are removed and inserted, which imposes a constant run-time load to the renderer. Also triangle strips must be gathered at run-time [Hop97] from individual triangles that may show up in an incoherent fashion somewhere on the mesh, wherever a vertex happens to be ‘unfolded’. Unlike in the mid-90s, current³ graphics hardware renders triangles very fast compared to the CPU speed, so that to bother with individual triangles becomes inefficient at some point.

- **No real-time mesh modifications.**

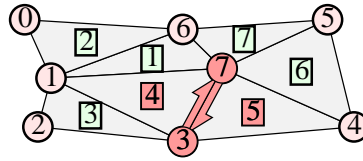
The simplification works offline, as a pre-processing step, to turn a high-resolution mesh into a multi-resolution mesh. When the input mesh is changed, just to re-run the simplification on the changed high-resolution parts is not enough: Simplification is a hierarchical process, and since it works by combining vertices, the changes are propagated also to un-changed surface regions, especially at lower resolutions.

- **No ‘real’ compression.**

The PM representation may be smaller in size than the input mesh, but it is still in the same order of complexity. The same is true for other mesh compression methods: They spend a very small, but asymptotically constant, number of bits per vertex to encode a mesh. In the ionic column example, however, a compression rate beyond that could be achieved if all the ridges would exist only once in a file, plus the information where the object has ridges.

To summarize, individual triangles are at a very low level of abstraction. A reasonable alternative, and one that also resolves some of these issues, is to group triangles together so that they form surface patches. With per-patch LOD adjustment, for instance, it is possible to switch between different resolutions of a whole group of triangles at a time.

³in 2004



edge	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
vertex	1	0	2	1	3	2	4	3	5	4	6	5	0	6	6	1	1	3	7	1	7	3	7	4	7	5	7	6
(to)	0	1	1	2	2	3	3	4	4	5	5	6	6	0	1	6	3	1	1	7	3	7	4	7	5	7	6	7
face	0	2	0	3	0	3	0	5	0	6	0	7	0	2	1	2	4	3	4	1	5	4	6	5	7	6	1	7
next	12	15	0	5	2	17	4	23	6	25	8	27	10	1	19	13	21	3	16	26	7	18	9	20	11	22	14	24

face	face 0						face 1			face 2			face 3			face 4			face 5			face 6		face 7				
vertices	0	6	5	4	3	2	1	7	6	1	1	6	0	3	1	2	3	7	1	4	7	3	5	7	4	6	7	5
edges	12	10	8	6	4	2	0	26	14	19	15	13	1	17	3	5	21	18	16	23	20	7	25	22	9	27	24	11

Figure 4.17: B-rep connectivity example. Upper table: The connectivity is defined by the halfedges, consecutive halfedges $2i$ and $2i + 1$ form one edge. Each halfedge has three references, the (to) row just list the mate vertex to increase readability. Lower table: Halfedges sorted according to face boundaries. Face 0 is the backface.

4.2 Boundary Representations and B-Rep Meshes

The term *boundary representation* is the name of the general approach to represent a real-world 3D object, or a 3D solid, in a digital computer solely by representing its surface. A 3D solid is bounded by a closed manifold surface, and this surface can be divided into discrete sets of 2-, 1-, and 0-cells. A variety of examples was given in chapter 2. The canonical, genuine domain of boundary representations is the class of closed manifold 2-complexes.

This term, and especially its shorter form *B-rep*, has also a more concrete, technical connotation. It denotes a class of data structures, a particular form of meshes with faces that can have any degree and possibly also holes. So they are a generalization of triangle meshes, but not only with respect to the face degree. B-reps can also realize more general embeddings, which makes them a data structure for representing patch complexes (Def. 2.34 in sec. 2.4). It is important to note that there is no such thing as ‘the’ B-rep data structure. B-reps can be implemented in many different ways. The reason is the broad applicability of the general B-rep approach.

A classical, important application domain for B-reps is *computer aided design*, CAD, where the surface of the objects is composed of trimmed B-spline or NURBS patches. Every NURBS patch is glued to other patches, with special continuity conditions along the borders. B-reps help to maintain the consistency of the CAD solids by storing their connectivity: They can keep patch references in their faces, and references to trim curves in the edges, for example.

Half-edge data structure. In the mid-80s quite a few different data structures for B-reps have been developed, besides the half-edge data structure most notably Baumgart’s winged edge data structure [Bau75], and the quad-edge data structure from Guibas and Stolfi [GS85]. They have different design tradeoffs, but they are all theoretically more or less equivalent as nicely summarized, e.g., by Lutz Kettner [Ket99].

For the same reasons as outlined for the *directed edge*-implementation in 4.1.4, a half-edge data structure was chosen. Another reason for this choice was compatibility: It seems that the half-edge data structure is today the most widely used B-rep data structure. This may be due to the fact that halfedges are quite useful and convenient to use as mesh iterators (sec. 4.1.3): A halfedge uniquely identifies a position on the mesh, i.e., one (vertex,edge,face) combination. With halfedge objects, a whole combination can be uniquely identified with a single pointer, the pointer to the respective halfedge object.

Which references to store in a halfedge? The halfedge data structure is an edge-centered data structure, i.e., the connectivity is entirely defined by the edges. The goal is to permit quick access to all incident entities, ideally in constant time, with a minimal space overhead in the data structure. Indexed face sets have one reference per halfedge, a vertex index. Directed edges need only two and can grant constant time access: *faceCCW*, *faceCW*, *vertexCCW*, *vertexCW*, and *mate* all have complexity $O(1)$. On the other hand, the maximum number of references is five: To the vertex and face, to the mate, and to the next and previous halfedges along the face boundary.

Since B-rep faces can have arbitrary degree it is not possible to group any fixed number of halfedges together, like three for the *directed edge* triangle meshes. Consequently, at least one more reference must be spent to connect the edges of a boundary cycle (without a proof). With only one reference, the face boundary is only a singly connected cyclic list. The consequence is that not all access functions have constant complexity any more – in one direction, the access time

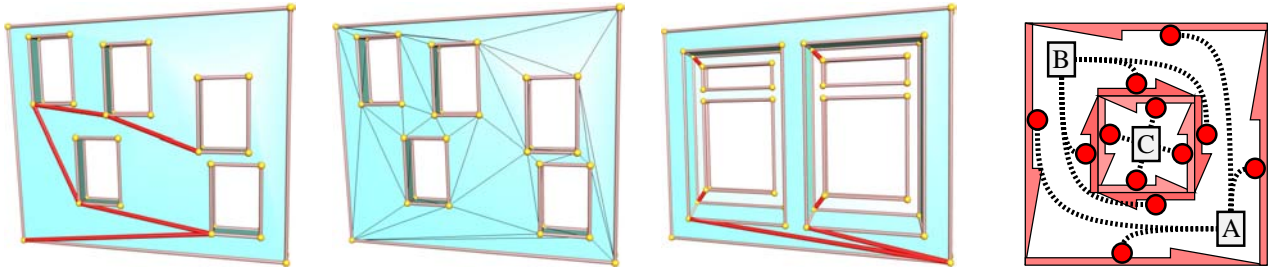


Figure 4.18: Red artifact edges to split up rings. When all rings are connected to the outer boundary, the face becomes a 2-cell (a). Artifact edges are somewhat arbitrary, it should be the job of the triangulation to correctly partition the face (b). Practically, this is annoying for instance when setting windows into a building façade (c). (d): Faces (*A* and *C*) and rings (*B*) are represented by the same C++ class `Face` that references one boundary cycle. But faces and rings know each other: $A.\text{nextring} = B$, $B.\text{nextring} = \text{NULL}$, and $B.\text{baseface} = A.\text{baseface} = A$.

is valence dependent (see `faceCW` code in Fig. 4.20 (a)). As it is no longer possible to derive the face index from the halfedge index a face reference is also mandatory. This is very redundant, indeed, as all face references in a boundary cycle are identical.

For B-reps, both face degrees and vertex valences are variable. The only constant valence incidence remaining is that in manifold surfaces, halfedges have only a single mate. This can be exploited to make an explicit mate pointer redundant, trading speed for space: The idea is to store pairs of halfedges together, i.e., both halfedges that make up an edge are stored in consecutive array positions. So in total, halfedges contain three references: To one vertex, one face, and to the next edge in the boundary cycle. As one example the familiar triangle mesh is listed in Fig. 4.17. The only higher degree face is the backface. The backface is not mandatory, the halfedges could equally point to `NULL` to flag a border. But note that due to the halfedge pairing, the halfedges on the other side of the the border are inevitable.

Duality: Face and vertex classes are (almost) symmetric. Note that halfedges connect the face boundaries as well as the edge cycles around vertices. Vertices and faces provide a single edge reference, just to make sure they are not a dead end for navigation. The roles of vertices and faces are symmetric and, in fact, exchangeable. Consequently, the dual of the abstract complex is directly accessible from the (primal) B-rep.

This symmetry is broken only by the fact that faces can have rings. As explained in sec. 2.2.4 (`makeEF`), this means that the dual graph can have complex vertices. Equivalently, to restore the symmetry of the data structure, vertices might also be allowed to be incident to more than one edge cycle. This would permit to represent also non-manifold vertices.

Faces can have rings. A design decision is that faces can have holes, i.e., multiple borders. The original motivation for rings is *trimming*: With trim curves, arbitrarily shaped holes can be cut out of the surface. It is extremely convenient when openings can be inserted into, e.g., a NURBS patch without having to split the patch up, which can be a complicated operation. When rings are not available, *artifact edges* have to be inserted. Artifact entities are entities that do not have to be inserted because of the logic of the model, but only because of technical requirements – or deficiencies.

In the spirit of patch complexes, B-rep faces are a device for grouping many triangles, or low-level cells, together that make up a surface. Triangles are required for technical reasons, but they should be generated automatically. The triangulation of a face with holes is shown in 4.18 (b): These triangles are only derived from another model representation that is adapted to the requirements of modeling, and to the inner logic of the model.

Without rings, interior loops have to be connected to the outer face boundary – but the choice, *which* pair of vertices is to be connected, is completely arbitrary. There is nothing special with the pair of vertices, e.g., in step 8 from Fig. 2.15. This is even worse when faces have multiple openings, as in Fig. 4.18. In order to remove all rings, and to make the face homeomorphic to the open 2-disc, all rings must be connected with non-intersecting paths to the outer face boundary. This introduces many edges that are incident to the same face on both sides. – Remind that with the Euler operators from section 2.2 an arbitrarily shaped opening in a polygonal face can be created with a single `killFmakeRH`.

To integrate rings into the data structure, note that faces reference only a single boundary cycle so far. This can be elegantly resolved by using the same class for faces and rings. This class is extended so that a face can reference another face, which is then a ring. In Fig. 4.18 (d), the neighbour of baseface *C* is ring *B*, whose baseface is *A*.

Implementation. The B-rep data structure is presented in Fig. 4.19. It is no surprise that it is based on a trait (from section 4.1.2). Some of its features and data structure design decisions are summarized in the table in Fig. 4.21.


```

struct Vertex : public V
{
    Vertex() { }
    Vertex(const V& data) : V(data) { }

    int& deadID    () { return status; }
    bool active    () { return status>=0; }
    int status;

    Edge* oneEdge;
};

struct Edge : public E
{
    Edge() { }
    Edge(const E& data) : E(data) { }

    Edge* faceCW    ();
    Edge* faceCCW   () { return next; }
    Edge* vertexCW  () { return mate()->next; }
    Edge* vertexCCW ();
    Edge* mate      () { return ((status&1)==0
                               ? this+1 : this-1; }

    int& deadID    () { return status; }
    bool active    () { return status>=0; }
    int status;

    Vertex* vertex;
    Edge* next;
    Face* face;
};

struct Face : public F
{
    Face() { }
    Face(const F& data) : F(data) { }

    bool isBaseface () { return baseface==this; }
    bool hasRings   () { return nextring!=NULL; }

    int& deadID    () { return (int&)oneEdge; }
    bool active    () { return baseface!=NULL; }

    Edge* oneEdge;
    Face* nextring;
    Face* baseface;
};

template<class Trait>
struct BRepMesh : public Trait
{
    typedef BRepMesh<Trait> Mesh;
    typedef typename Trait::V V;
    typedef typename Trait::E E;
    typedef typename Trait::F F;

    struct Vertex { ... see code to the left }
    struct Edge { ... see code to the left }
    struct Face { ... see code to the left }

    BRepMesh    () { }
    ~BRepMesh   () { }

    // Skipvector functionality
    void clear    ();
    bool reserve ( int kv, int ke, int kf );
    void purge   ( int* cv, int* ce, int* cf );
    bool relocate ();

    Vertex* newVertex ( const V& v );
    Edge* newEdges    ( const E& e0, const E& e1 );
    Face* newFace     ( const F& f );
    void deleteVertex ( Vertex* v );
    void deleteEdges  ( Edge* e );
    void deleteFace   ( Face* f );

    Skipvector<Face> faces;
    Skipvector<Edge> edges;
    Skipvector<Vertex> vertices;
};

struct Trait_VpFntc
{
    struct V { Vec3f position; };
    struct E { };
    struct F { Vec3f normal;
              int triChunk; };

    void triangulate ();
    void render      ();

    Skipchunk<GLuint> triangles;
};

template struct BRepMesh<Trait_VpFntc>;

```

Figure 4.19: B-rep mesh as a container. The entity classes (left) are local classes of the B-rep class (right). The B-rep uses the Skipvector container, the respective methods are explained in sec. 4.2.1. For newEdges and deleteEdges see Fig. 4.20 below. The trait adds to the container data structure the functionality to triangulate and render the faces, which is further explained in 4.2.2.

```

Edge* Edge::faceCW    ()      Edge* Mesh::newEdges(const E& e0,   void Mesh::deleteEdges(Edge* e)
{                               const E& e1)   {
    Edge* e1 = (Edge*) this;    {
    Edge* e0;                    Edge* e = edges.activate(e0);
    do {                          edges.activate(e1)->status = 1;
        e0=e1;                    return e;
        e1=e1->next;              }
    } while(e1!=this);
    return e0;
}

if ((e->status&1) == 0) {
    edges.deactivate(e+1);
    edges.deactivate(e);
} else {
    edges.deactivate(e);
    edges.deactivate(e-1);
}

```

Figure 4.20: Some B-rep functions. (a): Since edges have no pointer to the previous boundary halfedge, *faceCW* must be realized with $(m-1) \times \text{faceCCW}$. The *vertexCCW* navigation function is very similar. (b, c): Halfedges $(i, i+1)$ are activated as pair, and must be deactivated in reversed order $(i+1, i)$, so that they can be recycled.

1. **Entities are derived from their attributes.** The reason for this has already been discussed in sec. 4.1.2.
2. **References are pointers.** Besides the pros and cons from sec. 4.1.2, C++ path expressions have proven to greatly improve the readability of the source code, which makes it easier to maintain, and more robust. Second, mesh entities can act autonomously, and many functions, e.g., for finding all edges of a connected component, do not need access to a mesh, but only a halfedge to start from. Path expressions are also possible with index references (see code in Fig. 4.7), but then look like this: `edge→faceCCW(mesh)→mate(mesh)→faceCCW(mesh)`
3. **Incidence conventions.** Vertices and faces have only a pointer to one arbitrary incident halfedge: Faces to one boundary halfedge, vertices to one outgoing halfedge. A halfedge points to the vertex it emanates from, to the face *to its left*, and to the CCW next edge on the face boundary (CW next for rings).
4. **Face boundaries are singly connected.** It may be theoretically appealing that faceCW and vertexCCW are constant time face boundaries are doubly linked (otherwise, see faceCW code in 4.20 a). But adding one Edge::prev reference to each halfedge is quite a bit of overhead, since $|Halfedges| \approx 2 \cdot (|V| + |F|)$. And in practice the benefit is only small, since average face degrees use to be not very high. Second, it is remarkably seldom that there is no alternative to using faceCW and vertexCCW. In most cases, faceCCW and vertexCW can be used instead.
5. **No mate pointer: Halfedges are allocated in pairs.** Halfedges at consecutive array positions form one mesh edge. Let i be the index of a halfedge. If i is even, its mate has index $i + 1$, and $i - 1$ if i is odd. The mate can only be determined with access to the mesh: `mate = (edge - mesh.edges.begin()) &1 == 1 ? edge+1 : edge-1;`
Consequently, the Edge::mate function, and also Edge::vertexCCW, need to get a mesh pointer. – As an alternative, one bit of the status field is used to distinguish between odd and even indices, which is explained in section 4.2.1.
6. **Faces can have rings.** A Face object is either a *baseface*, or a *ring*. In either case it references only one halfedge, from a single boundary cycle. The edge cycle of all rings is CW oriented, whereas the basefaces are CCW. This is consistent in that the face interior is always to the left when walking along the boundary.
7. **Faces contain explicit ring pointers.** A face is a baseface if its baseface pointer points to itself. A singly connected list of rings is realized by the nextring pointer, which is NULL for the last ring – or for the baseface, if it has no rings.

Most faces do not have rings, and to spend two references for potential rings is much (8 bytes on 32 bit). Alternatively, the status field could be used to flag that a face has rings, or is a ring. The respective ring references would then have to be stored in a separate list.
8. **Navigation functions.** Since halfedges are autonomous objects, the navigation functions can be part of the halfedge class. From a single halfedge object, the whole connected component is reachable by just using the five halfedge navigation functions, together with Face::nextring and Face::baseface.
9. **Faces have no Vertex pointer.** The minimal genus 0-object is the pointed sphere (Fig. 2.2), which consists of only one vertex and one face – but no edges. Yet with an edge-based data structure, there is no connection between vertices and faces without edges. So due to a lack of references, neither a face nor a ring containing just an unconnected vertex can be represented.

A direct face-vertex connection could be realized by using a status bit flag: When the flag is on, the Vertex::oneEdge field, or Face::oneEdge, points not to an edge, but to a face, or a vertex, respectively.
10. **No NULL pointers.** All connectivity pointers must target valid entities. Boundaries can be realized only as boundary components, i.e., by inserting dummy faces (*hollows*). In particular there may be no isolated faces or vertices, i.e., the oneEdge must always point to a valid halfedge.
11. **No shell maintenance.** Arbitrarily many solid objects can be stored in the same mesh. There is no explicit list of connected components. One reason is that on the level of local mesh modifications, it is not possible to determine in constant time whether a certain operation finally cuts an object apart. Second, there is no 1:1 relation between shells and objects. Object administration is the job of higher software layers. It is often not even clear what an ‘object’ is.

Figure 4.21: Design decisions of class BRepMesh. This is a summary of the essential options and alternatives, and the decisions that have been made. The result is a set of conventions to decide whether a mesh is valid.

4.2.1 Skipvectors and Skipchunks

Modeling implies that objects can be changed. Changeable meshes have also many applications beyond modeling, just to mention level-of-detail and view-dependent refinement. It is crucial for the performance to have low-level data structures that efficiently support quickly varying data. In the case of meshes, it must be possible to add and remove lots of mesh entities on a per-frame basis. The code in Fig. 4.19 uses a new data structure for vertex, edge, and face lists, the skipvector.

A *skipvector* is a container data structure that contains a sequence of items, that is (a) organized as an array, but (b) supports quick insertion and deletion. It therefore combines the advantages of arrays, the direct random access to elements, and of lists, namely flexible insertion and deletion. – To achieve the full performance with skipvectors, it is necessary to know the mechanism behind. The implementation is not completely hidden away, like with STL’s vector or list containers.

Arrays versus lists. The STL specification [SL95] says that the vector container is for static, and list for dynamic data. The vector permits $O(1)$ insertion and deletion only at the end. Deleting other objects has linear cost, since the gap in the array must be closed. The obvious implementation of a vector is as a C++ array.

The list container permits $O(1)$ insertion and deletion everywhere but, other than the vector, it does not permit random access: The k th element can only be accessed by $(k - 1)$ -times stepwise iteration, starting from the beginning. The obvious implementation of a list is as a doubly-linked list, with two references *prev*, *next* per element. The danger when using lists is *memory fragmentation*: After many insertions and deletions it may be that consecutive list elements reside in distant locations in memory. list implementations exist that by default allocate each single element individually using *new*. This fragmentation kills the cache coherence of the CPU, and can impair performance drastically. Tests have shown that the time needed to simply iterate over a list with n elements can increase by a factor of 10 after many random insertions and deletions – and this factor can even (discontinuously) increase with greater n .

The conclusion is to use arrays whenever possible, since they guarantee that all objects reside in a coherent chunk of memory – were there only not the deletion issue.

Skipvectors are almost vectors. A skipvector maintains an array of objects of a given parameter type T . The deletion issue is resolved by switching elements off, instead of actually deleting them. This leads to an unusual requirement for the containee, class T , of a `Skipvector<T>`. Each element of type T must provide the following functions:

- `T::T()` the standard constructor for initialization,
- `T::~~T()` the standard destructor,
- `int& T::deadID()` a place to store an integer, and
- `bool T::active() const` a user-defined flag to tell whether an object is active or inactive.

In case an object is to be deleted from the array, this element has to offer a place where the skipvector can store a negative integer. What for? The skipvector maintains a reference, an index i , of the element that was last recently deleted. In case another element j is deleted, $(-2 - i)$ is stored in the `deadID()` of j , and j now becomes the skipvector’s last recently deleted object. No destructor is called for i , or j , so all deleted elements are actually still ‘alive’, but have become inactive. It is in the responsibility of the programmer to make active return `false` when an object is deactivated.

In case a new object is needed, the last recently deleted object j is recycled. That is, the skipvector looks at item j and sets $k = \text{item}[j].\text{deadID}()$ and now takes $(-2 - k)$ as last recently deleted object. It then stores the value 0 in `deadID()` of j , and eventually returns a pointer to j for its re-use. No constructor is called. It is in the responsibility of the user to properly initialize the object. A skipvector can therefore be seen as an ‘array with gaps’. When looping over the array elements, inactive elements need to be skipped (hence the name). An additional if-statement in the loop body is the price to pay for the ability to remove elements from an array in constant time:

```
for (Face* face = mesh.faces.begin(); face != mesh.faces.end(); ++face) {
    if (face->active()) { render(face); }
}
```

Garbage collection. It may happen that loops become very inefficient when relatively few elements are sparsely distributed in a big array. To find out whether the majority of all elements is deactivated, the size of the active elements can be compared against the range that the elements span, which equals $(\text{end} - \text{begin})$. The skipvector provides a garbage collection, the *purge* function, that can be applied when for instance $\text{range} > 3 \cdot \text{size}$.

The *purge* function moves all active items to the beginning of the array, removing all gaps between them, and moves all inactive items behind them. It uses `memcpy` from the standard C library to do that. Thereby, it changes the location of items in the array, a situation similar to the relocation problem: Pointers on array elements must be updated. Since items do not leave the array, the update can be done by adding an integer offset to existing pointers. The *purge* stores the

```

template<class T>
struct Skipvector
{
    Skipvector ();
    ~Skipvector ();

    T* activate ();
    T* activate (const T&);
    void deactivate (T* item);

    T* begin ();
    T* end ();
    T*& oldBegin ();
    T* operator() (int index);
    T& operator[] (int index);

    bool clear ();
    bool reserve (int n);
    bool purge (int* c);
    void relocate (T*& t);
    int size ();
    int range ();
    int capacity ();
};

template<class T>
struct Skipchunk
{
    Skipchunk ();
    ~Skipchunk ();

    int activate (int size=-1);
    T* addToChunk (int size);
    void deactivate (int id);

    T* chunkBegin (int id)
    T* chunkEnd (int id)
    int size (int id)
    int start (int id)

    void clear ();
    bool reserve (int n);
    void purge (int* idc);

    int size ();
    int range ();
    int capacity ();
};

void BRepMesh::purge(int* cv,
                    int* ce, int* cf)
{
    vertices.purge(cv);
    edges.purge(ce);
    faces.purge(cf);
    for(Vertex* v =vertices.begin();
        v!=vertices.end(); ++v){
        edges.relocate(v→oneEdge);
    }
    for(Edge* e =edges.begin();
        e!=edges.end(); ++e) {
        vertices.relocate(e→vertex);
        edges.relocate(e→next);
        faces.relocate(e→face);
    }
    for(Face* f =faces.begin();
        f!=faces.end(); ++f) {
        edges.relocate(f→oneEdge);
        faces.relocate(f→nextring);
        faces.relocate(f→baseface);
    }
}

```

Figure 4.22: Skipvector and Skipchunk interfaces. Right: Implementation of the purge routine from Fig. 4.19.

individual offset values into an integer array c , whose size must be at least the skipvector range. If t is a pointer into the array, then t can be corrected with `if (t !=NULL) { t += c[t-begin()]; }`

The skipvector provides the `relocate` method to do this correction. As an example, Fig. 4.22 (c) shows the garbage collection of mesh entities. This is the implementation of `BRepMesh::purge` (Fig. 4.19 c), which is very similar to the relocation repair. Note that (a) the correction arrays are expected to be allocated outside of `BRepMesh::purge`, and (b) loops no longer have to test the active flag after purging the skipvector (until the next deletion).

The cost of purging an array is linear in its range, each element must be memcopied at most twice. It is worthwhile when, after a sequence of insertions and deletions, many loops have to be made over the array.

Memory allocation strategy and relocations. When more objects are needed than were deleted, the skipvector follows the same strategy as the STL vector: The array is dynamically resized to twice its size (using `realloc` from the standard C library), which may lead to a relocation. As discussed in sec. 4.1.2, the problem is that pointers on array elements must be updated. This is to be done by a *user-defined relocation repair*, similar to the function in Fig. 4.5. The code may be simpler though, in the fashion of Fig. 4.22 (c), because the `relocate` function also serves for relocation repair. A relocation has happened when `oldBegin() !=NULL`. The `oldBegin` must be set to `NULL` after the relocation has been properly repaired.

A relocation may happen with `activate` and `reserve`, since they increase the size, and with `purge`, which cleans up unused items. In case of a purge relocation, the purge repair can be safely performed *after* the relocation repair. The `reserve` function should be used whenever the number of objects to be inserted is known in advance. The number may be roughly, but conservatively, approximated. After a sufficiently large reserve no relocations can happen any more, and no relocation checks are necessary after `activate` etc.

The `deactivate` method only switches items off, and `clear` just sets `end` to `begin`. Both functions affects the size and possibly the range, but not the capacity. The only way to reduce the capacity, i.e., the memory consumption, is to purge the skipvector. After a purge, the capacity is the next power of two greater than the size. It is vital that in case items of type T contain dynamic data, i.e., when each vertex contains for instance a set, these data are released before either `deactivate` or `clear`. To execute, e.g., `T::clear`, if it exists, is in the responsibility of the user.

Skipchunks. It happens that it must be possible to dynamically allocate and delete not only individual data elements, but whole arrays of data elements. A notable example is polygon triangulation: When the polygon is edited, the list of triangles must be updated, e.g., deleted and replaced.

The skipchunk (see Fig. 4.22 b) is a container data structure that permits to request whole arrays of elements of a given type T . The `activate` method returns the ID of the chunk as an integer. This ID can be used to traverse the chunk, get its size, etc. The last activated chunk can also be expanded using `addToChunk`. Note that the purge routine returns chunk ID offsets, not pointer offsets. There is no functionality to iterate over all chunks.

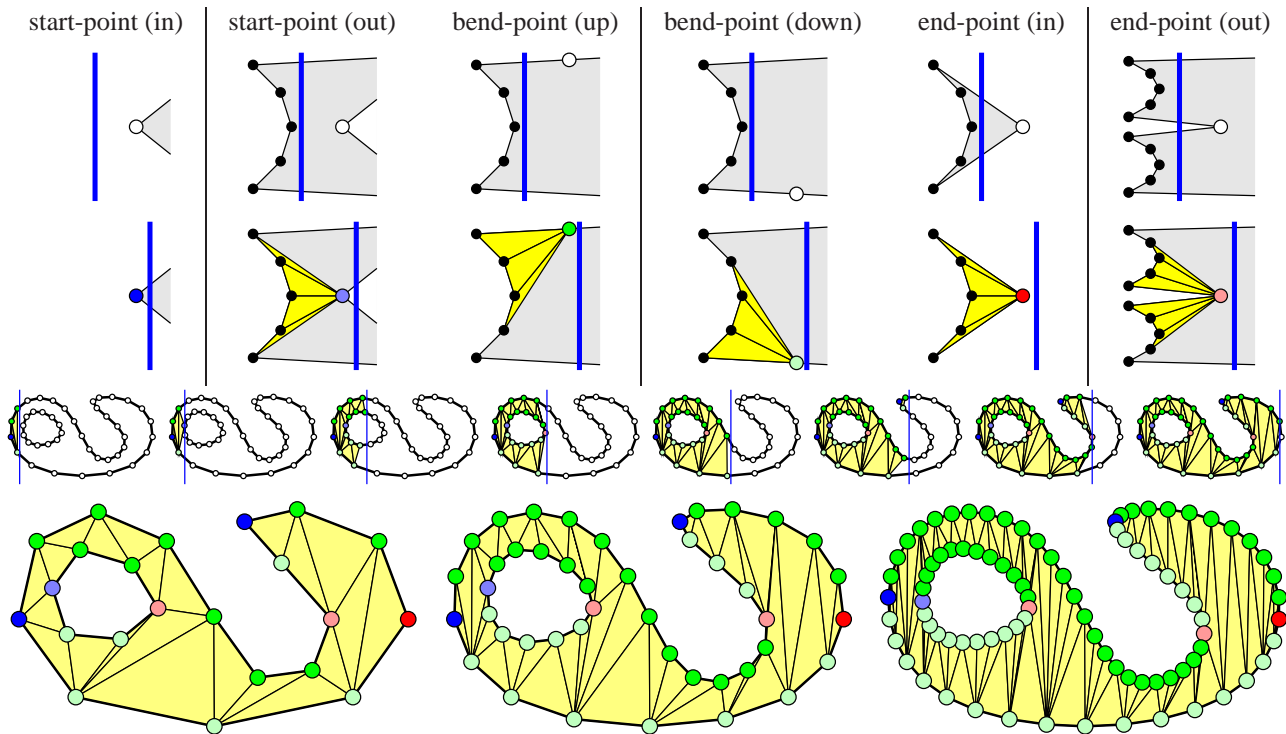


Figure 4.23: Sweep-line triangulation from Mehlhorn. Top rows: Situations before and after the sweepline passes. The un-triangulated interior is grey. The algorithm maintains a list of ‘pockets’ sorted in vertical direction. A start-node opens a pocket or splits one in two, an end-node closes or unites different pockets. As many triangles as possible are added in every step, so that the pockets are always concave. Bottom rows: Irrespective of the sampling density, the boundary polygon has 6 critical points (3 start-, 3 end-points).

4.2.2 Polygon Triangulation

The *linear embedding* of a B-rep is an embedding such that edges are straight line segments, and all edges and vertices that belong to a face are contained within the same plane. So each face of a linear B-rep is an arbitrarily shaped polygon, possibly with holes, floating in 3-space. In order to display it using a low-level graphics API, it must be tessellated, i.e., partitioned into triangles. Polygon triangulation is not a very complex problem, and a variety of different efficient methods exist. This is also indicated by the fact that the size of the output, the number of triangles, basically equals the input size, the number of vertices and holes.

Theorem 4.5 (Size of a triangulation)

Any triangulation of a polygon with n vertices and r rings contains $t = v + 2r - 2$ triangles.

To prove this, the triangulated polygon can be considered as a connected planar 2-complex. This complex contains $f = t + r + 1$ faces: besides the triangles also one face for each ring, and one surrounding unbounded or backface. The idea now is to count the edges twice by counting the triangles. This yields $2e = 3t + v$, since by counting three edges per triangle, all interior edges of the triangulation are counted twice; but not so the edges of the polygon. But irrespective of the number of holes, every polygon with v points also has v edges, since one edge belongs to every vertex. On the other hand, the Euler-Poincaré equation says that $v - e + f = 2 \Leftrightarrow 2v + 2f - 4 = 2e$. Since $f = t + r + 1$, this equals $2v + 2(t + r + 1) - 4 = 2e = 3t + v$, which proves the theorem.

A sweepline algorithm. Computational geometry offers triangulation algorithms that use a partition into monotone pieces [dvOS97], or very efficient randomized methods such as the $O(n \log^* n)$ algorithm from Raimund Seidel [Sei91]. Another particularly useful paradigm from computational geometry is the class of *sweepline algorithms*. The idea is to sort all input vertices, e.g., from left to right, and to process them one by one. This can be imagined as a continuously moving vertical line, the *sweep line*. Mehlhorn has presented a sweepline algorithm for polygon triangulation [Meh84], which is illustrated in Fig. 4.23: Every polygon vertex is an event point, and whenever the blue sweepline passes a vertex, some triangles are added to the triangulation, according to the type of the event point.

```

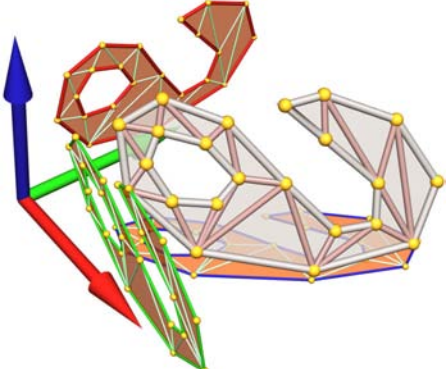
struct TriangulatePolygon
{
    typedef unsigned int ID;
    typedef vector<ID> IDs;
    struct Triangle { ID t[3]; };
    typedef vector<Triangle> Triangles;

    TriangulatePolygon ();
    ~TriangulatePolygon ();

    void setPointArray (void* points,
                      int stride);
    void beginFace (const Vec3f& n);
    IDs& points ();
    IDs& loops ();
    bool triangulate ();
    Triangles& triangles ();
};

nodes() 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
loops() 16 22

```



```

void Trait_VpFntc::triangulate () { ... (init)
    Face *face, *ring;
    Edge *edge, *edgeEnd;
    int i,k,n,c;
    TriangulatePolygon polytri;
    polytri.setPointArray(&vertices.begin()->position,
                          sizeof(Vertex));
    for(face=faces.begin(); face!=faces.end(); ++face) {
        if(face->active() && face->isBaseface()) {
            polytri.beginFace(face->normal);
            ring = face;
            i = 0;
            do {
                edge = ring->oneEdge;
                edgeEnd = edge;
                do {
                    i++;
                    k = edge - edges.begin();
                    polytri.nodes().push_back(k);
                    edge = edge->faceCCW();
                } while(edgeEnd!=edge);
                polytri.loops().push_back(i);
                ring = ring->nextring;
            } while(ring!=NULL);

            if(polytri.triangulate()) {
                n = 3 * polytri.triangles().size();
                c = triangles.activate(n);
                face->triChunk = c;
                memcpy(triangles.chunk(c),
                    &*polytri.triangles().begin(),
                    n*sizeof(ID));
            }
        }
    }
}

```

Figure 4.24: Triangulation API.

Care must be taken with vertical polygon segments, since both vertices pass the sweepline simultaneously. This case can be reduced to the standard case by ‘simulation of simplicity’: The sweepline is imagined to be slightly slanted to the left, so that from two vertices with equal x -coordinate, the vertex with smaller y -coordinate is processed first.

The triangle aspect ratio issue. There is a practical problem with this algorithm, and in fact with any triangulation algorithm that uses only polygon vertices: When the polygon boundary is densely sampled, many long and thin triangles are created, as in Fig 4.23, bottom right. Especially with specular materials, this gives visible artifacts when using Gouraud shading: the illumination is computed at the vertices, and only interpolated in the triangle interior. To resolve this issue, and to produce triangles with better aspect ratios, the triangulation must also use sample points from the polygon interior, e.g., the so-called Steiner points [Epp92].

On the other hand, the left-to-right order of the triangle creation is also a feature of this algorithm, since it can greatly speed-up the rendering. There are usually much fewer start- and end points than bend points. When triangulating, e.g., a convex shape, all except two points are bend points, and the triangles basically form a triangle strip. Today’s graphics hardware can exploit this in fact even when the triangles are not rendered using the triangle strip primitive (cf. Fig. 2.25 from sec. 2.3.2), but arrive as index triplets. The graphics hardware maintains a *vertex cache* [Hop99], so that recently processed (transformed, lit, etc.) vertices can be identified by their index and can, thus, be re-used.

To retain this feature, but still remedy the shading artifacts, another alternative is to use a shading model other than Gouraud shading. The solution today is to use pixel shaders [MTP*04].

The triangulation API. The actual triangulation algorithm can be completely hidden behind a suitable interface (API). Fig. 4.24 gives a very explicit code example that not only demonstrates the usage of a `TriangulatePolygon` object (highlighted in bold as `polytri`) – it also gives a practical example of boundary traversal using halfedge navigation.

Similar as with OpenGL vertex arrays, the polygon vertices from each face are fed into the triangulation algorithm only as indices. These indices directly refer to the vertex array of the B-rep, with the position field of the first vertex as



```

void Trait_VpFntc::render() { ... (init)
  glEnableClientState(GL_VERTEX_ARRAY);
  glVertexPointer(3, GL_FLOAT, sizeof(Vertex), &vertices.begin() -> position);
  for(Face* face = faces.begin(); face != faces.end(); ++face) {
    if(face -> active() && face -> isBaseface()) {
      glNormal3f(face -> normal.x, face -> normal.y, face -> normal.z);
      glDrawElements(GL_TRIANGLES, triangles.size(face -> triChunk),
                    GL_UNSIGNED_INT, triangles.chunk(face -> triChunk));
    }
  }
  glDisableClientState(GL_VERTEX_ARRAY);
}

```

Figure 4.25: Rendering a triangulated B-rep.

node with index 0. Since the skipvector holds the B-rep vertices in an array, the i -th next node can be found by adding an offset of $i \cdot \text{sizeof}(\text{Vertex})$ bytes, which gives the location of `mesh.vertices[i].position`. Similar to OpenGL's `glVertexPointer`, the `setPointArray` method expects a *stride*, the memory offset between consecutive vertices, or nodes.

The loop iterate over all B-rep basefaces. The body of the loop consists of two parts. The first part demonstrates the typical code to traverse all edges of a polygon with holes by using halfedge navigation functions. The face boundary is cyclic, so it is usually traversed using `do .. while` together with `faceCCW`, rather than in a `for`-loop. This inner loop is enclosed by another `do .. while` loop to iterate over all rings of the face (if it has any).

Projection to a principal plane. For each face, the triangulation is initialized with `polytri.beginFace(n)`. Using the normal vector n , one of the six *principal planes* is chosen, i.e., one of the xy , yz , zx , and xz , zy , yx planes. The face polygon lies on an arbitrary plane in 3D, the *face plane*; but triangulation is a 2D problem. Fortunately, the triangulation of a polygon is affinely invariant: The projected triangulation is also a triangulation of the projected polygon. As an example, the projections to principal planes of the example polygon are illustrated in the image in Fig. 4.24.

For numerical stability, the best principal plane is selected according to which of the normal coordinates (n_x, n_y, n_z) has the largest absolute value. The sign of this coordinate determines the plane orientation, to choose between xy or yx , etc. Note that the projection on the principal planes is special since it can be performed without *any* calculations, by just leaving one coordinate away.

Storage order and rendering. During traversal, the node indices are collected into one coherent `points()` array. In order to distinguish the different closed polygon loops, a second `loop()` array contains the index of the first node after the respective loop. This is shown in the table above the image in Fig. 4.24 for the example polygon with 22 nodes. When the nodes are in the same order as the point coordinate array, the node list is the identity. The loop list contains 16 and 22, for the lengths of outer boundary and ring, which are 16 and 6.

The result of calling the `triangulate()` method is a sequence of $n = v + 2r - 2$ consecutive index triplets, one for each triangle of the triangulation. They can be directly memcopied to a chunk of $3 \cdot n$ indices reserved in a skipchunk, namely the `Trait_VpFn::triangles` from Fig. 4.19. To later retrieve the chunk, its integer ID is stored in the face.

The net benefit of this effort is that rendering the triangulated B-rep becomes very easy. The code in Fig. 4.25 is extremely concise. Again the B-rep's vertex array is taken directly as the source for the point coordinates, using the same stride `sizeof(Vertex)`. All that needs to be done is to call `glDrawElements` for each face with the triangle indices stored in the triangle chunk of the face.

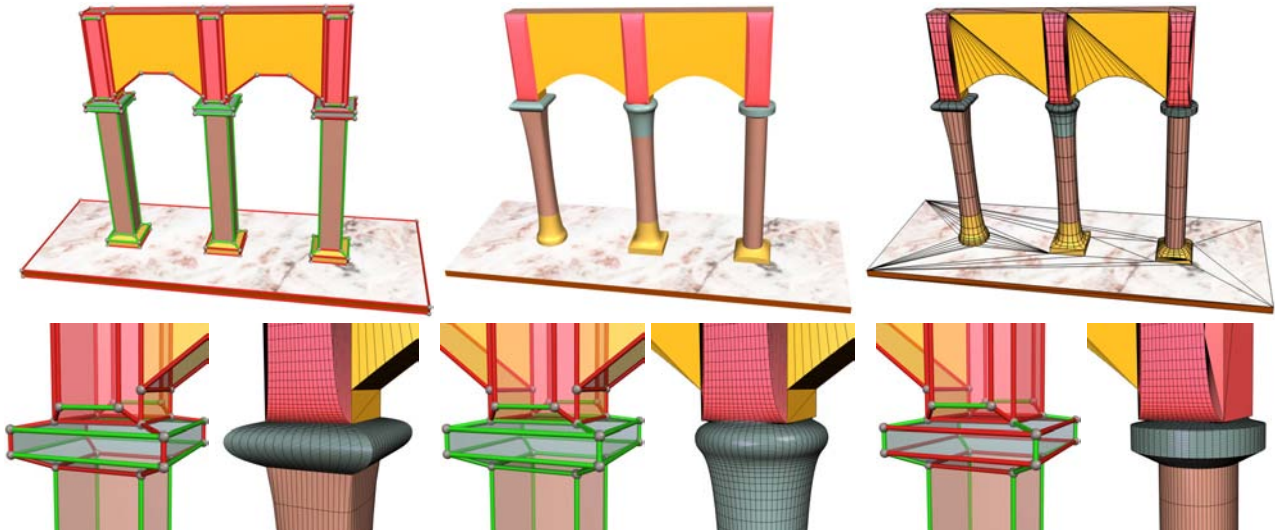


Figure 4.26: Combined B-rep Example: Arcade. Lower row: All column capitals have an identical control mesh, only the sharpness flags of the edges are different. A range of different shapes can be achieved by combining smooth transitions with sharp creases in various ways. Upper row: The polygonal control mesh has only 144 vertices and 126 faces, and also three rings. The bases of the column are rings of the floor, which is respected by the tessellation. This works also with the left column where the ring at the base is a B-spline curve (1c).

4.3 Combined B-Rep Meshes

The *combined B-rep*, short cB-rep, is a multiresolution data structure for interactive modeling and visualization of models composed of both free-form and polygonal parts. It is based on a half-edge data structure combined with Catmull/Clark subdivision surfaces, hence its name. Combined B-reps bridge the gap between polygonal models on the one hand, and free-form modeling on the other, and join both into one unified shape data structure.

This flexibility is achieved by attaching just one bit of information to every edge in the mesh, namely a *sharpness flag*, to distinguish between *sharp edges* (red) and *smooth edges* (green). In regions with only sharp edges, B-rep faces are rendered using standard polygon rendering, while in smooth regions the B-rep is regarded as a control mesh for Catmull/Clark surfaces to create smooth free-form shapes.

From a user's perspective, the great thing is that not too many degrees of freedom are added to the mesh: The shape of the object is completely determined by the vertex positions and the sharpness bits, and, of course, the mesh connectivity. Usually the shape of a model is just modified by moving control points, or mesh vertices – which is quite intuitive, but can be tedious. With combined B-reps, there is another, powerful way to modify the shape, by flipping the sharpness of some edges. This is in fact quite intuitive as well, an artist just has to keep in mind that

- for vertices, the number of incident *sharp* edges counts, and that
- all faces with a smooth edge are rendered as subdivision surfaces.

An example for the expressive power gained with this extension to the B-rep meshes from the last section is shown in Fig. 4.26. Combined B-reps allow to describe a complex shape very concisely: It is sufficient to basically just paint the edges of a mesh either green or red. But beyond just a more flexible and concise shape description, combined B-reps also offer two features that are indispensable when interactive shape design is the goal:

- **Interactive Visualization: View-dependent, adaptive level-of-detail**

A combined B-rep is a multi-resolution mesh. The free-form parts of the surface are partitioned into patches using the technique that was presented section 3.4. In every frame, the resolution of each patch is chosen so as to provide an optimal balance between display quality and rendering performance.

- **Interactive Modeling: The mesh can be changed at runtime**

The user is free to make arbitrary modifications to the mesh. Vertices can be moved around, the sharpness of any edge can be toggled, and the connectivity can be modified as well (but must remain manifold). The tessellation of changed surface parts is incrementally updated in real-time. By *tessellation on demand*, only the changed parts of the tessellation are re-computed, and only in the resolution that is needed – without any lengthy offline pre-processing.

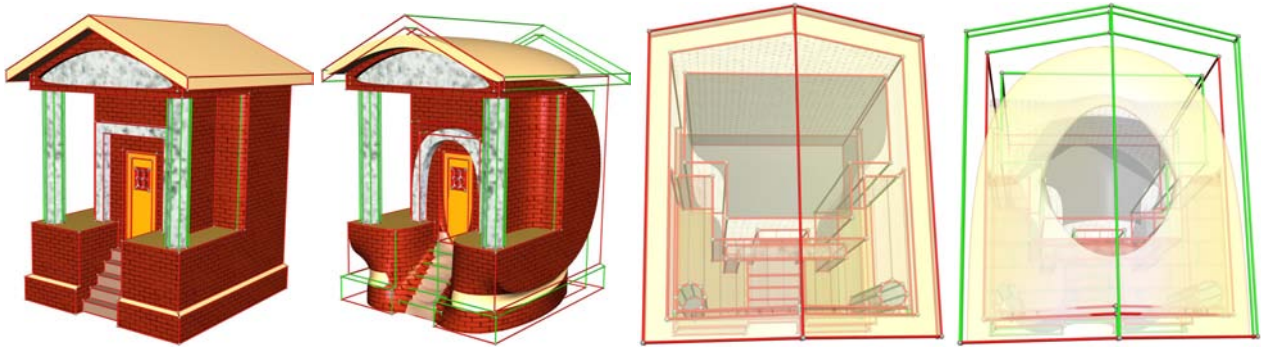


Figure 4.27: Combined B-rep Example: Temple. Changing the sharpness of edges can have drastic effects. The right temple is an attempt to smoothen as many edges as possible without introducing artifacts, like self-intersections or triangulation problems. The roof pulls itself together and forms a shape that looks like the head of a mushroom. The high-valence faces to the right and left sides of the steps very much attract the steps' surface.

4.3.1 Background on Combined B-Reps and Related Work

The initial motivation for the development of combined B-reps was the quest for a surface representation suitable for interactive modeling. The surface was supposed to render quickly, to have adequate – not too many and not too few – degrees of freedom, and, most importantly, should cope efficiently with incremental shape changes through some kind of selective update scheme.

When designing an object representation for 3D models, there is a certain trade-off between ease-of-manipulation and rendering efficiency. For many surface representations, e. g. NURBS, quite efficient adaptive rendering schemes have been developed, from Kumar's *torpedo room* [KML96] to the *fat borders* from Balázs et al. [BGK03]. Alternatively, surface models can be tessellated and represented by triangle soups, or using simplification and multi-resolution meshes. An obvious drawback of all these approaches is that if an object undergoes shape changes, the costly preprocessing has to be re-done. In all application areas where 3D objects are to be changed on-line, in unforeseeable ways, and based on user interaction, instant visual feedback is the key. Offline preprocessing is not an alternative then. Apparently, the only possible way out of this dilemma is

- to *intertwine* the preprocessing with the interactive display, and
- to design *update-able* data structures, which permit to selectively re-compute only parts of the pre-processing.

Unfortunately, the subject of suitable shape representations, especially for changeable shapes with both free-form and polygonal parts, has received relatively little attention so far as a subject in its own right.

Representation of deformable models. This question is only treated as a side-issue in the large body of literature on interactively deformable models. These approaches are all based on some kind of underlying shape representation that permits real-time manipulation, e.g., triangle meshes [WW94, Gai00, GD99], implicit surfaces [Baj96, BCX95, HQ01, DTG96, MCCH99], volumetric simplicial complexes [CFM*94], discrete levels of detail [DDCB01], subdivision solids [MQ02, McD03], or even point clouds [PKKG03a]. The focus of these papers, though, is in most cases the modeling functionality rather than the underlying shape infrastructure. The subject of incremental tessellation updates is only treated by Li and Lau [LL99] in greater detail, for the case of deforming NURBS surfaces.

One problem of NURBS, as well as Sederberg's non-uniform subdivision surfaces [SZSS98], is that they offer *too many* degrees of freedom. On the other hand, not all control points of a patch can be freely moved, there are invisible dependencies between some of them. With NURBS in particular there are the well-known problems of maintaining the geometric continuity with an irregular patch layout (see Farin's book [Far02]). This impairs their usability in interactive design, especially in comparison with subdivision surfaces. For the latter, practically instant feed-back can be guaranteed for interactive modifications involving several hundred control mesh faces, due to very fast tessellation algorithms.

For Catmull/Clark surfaces, Bolz and Schröder [BS02a] report rates of 5.5 million quads that can be generated per second using their tessellation scheme. This raises the question of whether caching the tessellation data is worthwhile altogether, as 180K quads can be created at 30 fps with this approach. But doing this imposes a 100% CPU load – while with the approach presented in section 3.4.3, *no* further computation is necessary for adaptive display, once the caches are filled. And the generation is only one component, equally important is adaptive display without cracks.

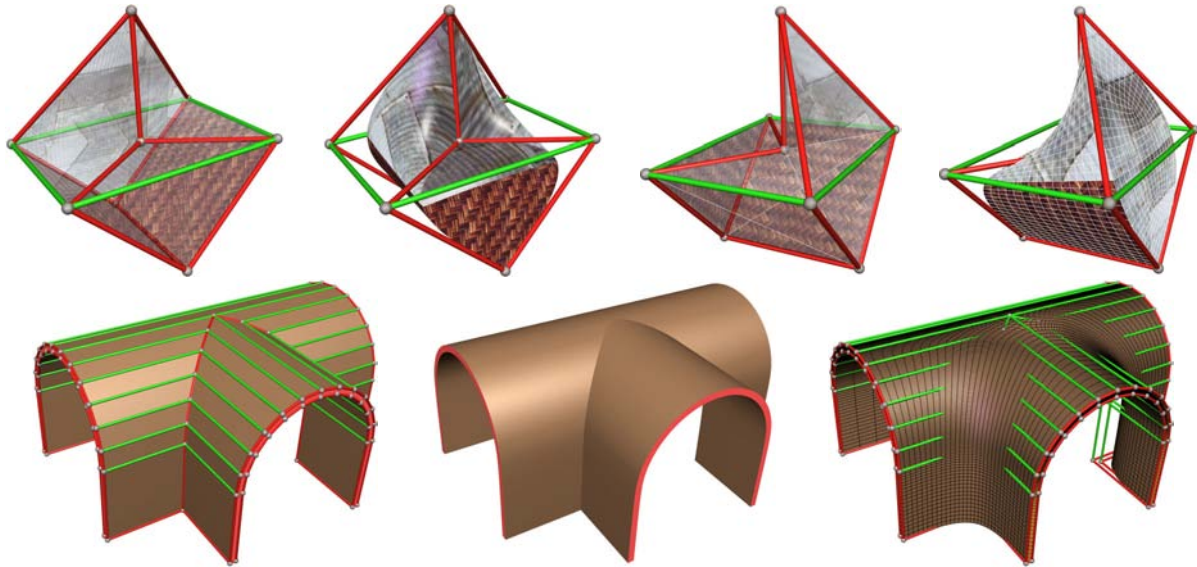


Figure 4.28: Combined B-rep Examples: Non-planar Faces and Tunnel. Top row: Subdivision surfaces are most useful with non-planar faces. The surface follows the control mesh in an intuitive way, other than the triangulation, which can break with weird, curved face boundaries that do not admit a projection without self-intersections. Bottom: The tunnel shows how easy it is to switch between a sharp, CSG-like joint, and a smooth blended joint. It is just a matter of whether the edges in the corner form a crease, or not.

Multi-resolution mesh editing. There is also a body of literature on editing multi-resolution triangle meshes, summarized e.g. in [KBB*00]. Using a *decomposition operator*, a fine-to-coarse hierarchy is established, and the shape can be edited at any level. Shape detail is transferred back on the shape using the inverse operator, e.g. by subdivision. The correspondence between different levels is maintained during modeling, either through a semi-regular connectivity (cf. [ZSS97]), or via local parametrizations like in [KBS00]. The combined B-rep approach is different since it restricts the modeling operations to the base mesh. The control mesh captures the complete shape information, there are no detail coefficients. The obvious drawback is that with ‘pure’ Catmull/Clark, there is no fine level feature editing; no editing of the basic shape while preserving high-frequency detail is possible.

A major problem when editing multi-resolution meshes is to keep the tessellation consistent. Approaches that maintain several different levels of detail explicitly have limitations with large-scale modifications, especially with genus changes. Cheng et al. present a quite interesting approach in [CDES01] for a consistent adaptive triangulation of a *skin surface*, basically an implicit surface derived from a set of moving weighted points, e.g., spheres. Their mesh update is based on local operations (collapse/split), but also has operations to change genus. This makes smooth transitions possible even between objects that differ in genus. Their approach could eventually be extended to produce a multi-resolution mesh, i. e., when the modifications are carried out simultaneously on different levels.

Advantages of the combined B-rep approach. But all approaches that explicitly manipulate low-level triangles, such as [GD99, Gai00], suffer from the fact that (i) to bother with individual triangles is inefficient when the graphics hardware can display them faster than the CPU processes them, and (ii) local modifications interfere with rendering optimizations and triangle strip generation. The combined B-rep approach does not have these problems: It is strictly top-down, on a per-patch per-face basis, where the Catmull/Clark surface is regular and the tessellation scheme can be highly optimized. Irregular cases are captured on an intermediate level, using the vertex and face rings from section 3.4.2. Second, the technique of multi-resolution rendering by patch sub-sampling from section 3.4.3 permits to pre-compute optimized triangle strips, avoiding cracks in the tessellation even with arbitrary depth differences of neighbouring faces.

The tessellation is considered a transient artifact which can be quickly (re-)generated on demand, and may also be deleted if no longer needed. From this point of view, B-rep edges are *feature edges*, which are distinguished from *artifact edges* in the face interior, generated automatically by triangulation or subdivision.

Combined B-reps in the architectural domain. The suitability of combined B-reps was demonstrated for the faithful representation of architectural features, e.g., in the VAST conference in Glyfada in [HF01]. The following Figures 4.29, 4.30, and 4.31 give an impression of their usefulness for this domain.



Figure 4.29: Combined B-rep example: Ornamental detail. In classical architecture, most often profiles are rounded only in *vertical* direction. This can be concisely expressed with combined B-reps by making paths of *horizontal* edges smooth. The side section, a sharp face, nicely shows the resulting B-spline control polygon.



Figure 4.30: Combined B-rep example: Levels of window detail. Architectural models typically exhibit rounded features only in few, but very distinct places. Efficient LOD control is vital for interactive inspection of interesting building details (c). But when the user just passes by a house with many windows, a low LOD is needed (a).

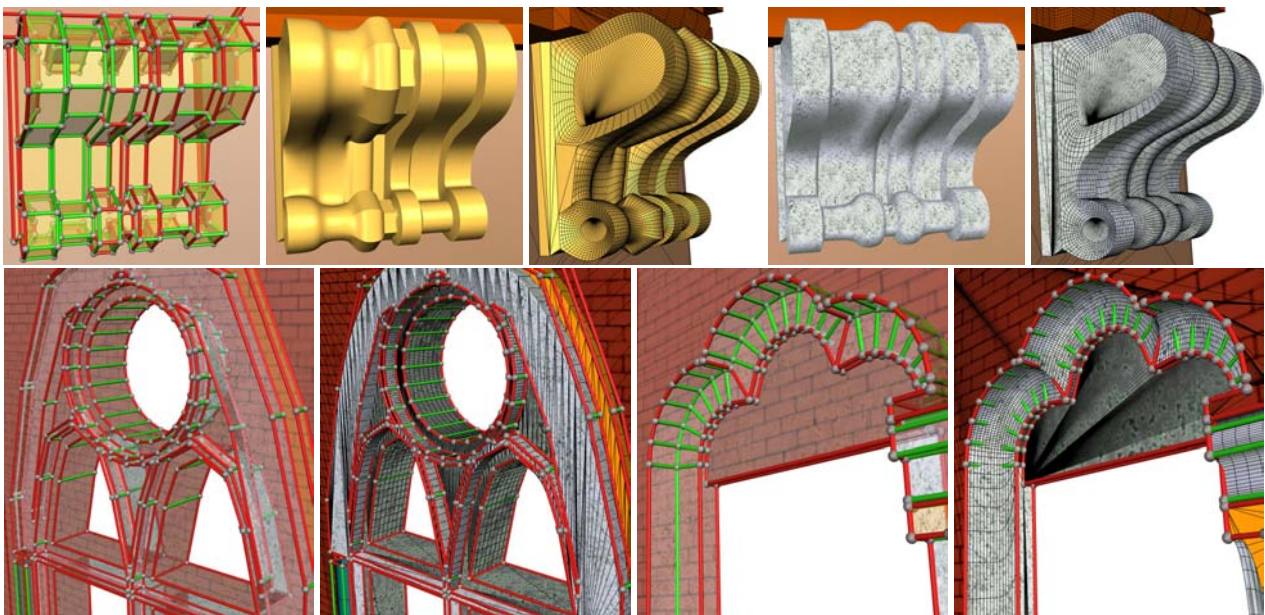


Figure 4.31: Combined B-rep example: Window decoration. Upper row: The lean model (1a) demonstrates smooth, horizontal sharp, and vertical sharp decorations. Sharp paths emphasize the ‘flow’ of an ornament (1d, e). Lower row: Very coarse, but regular models are sufficient to create very smooth, realistic architectural shapes. The crucial question always is: What is the ‘right’ control mesh for a particular type of construction?

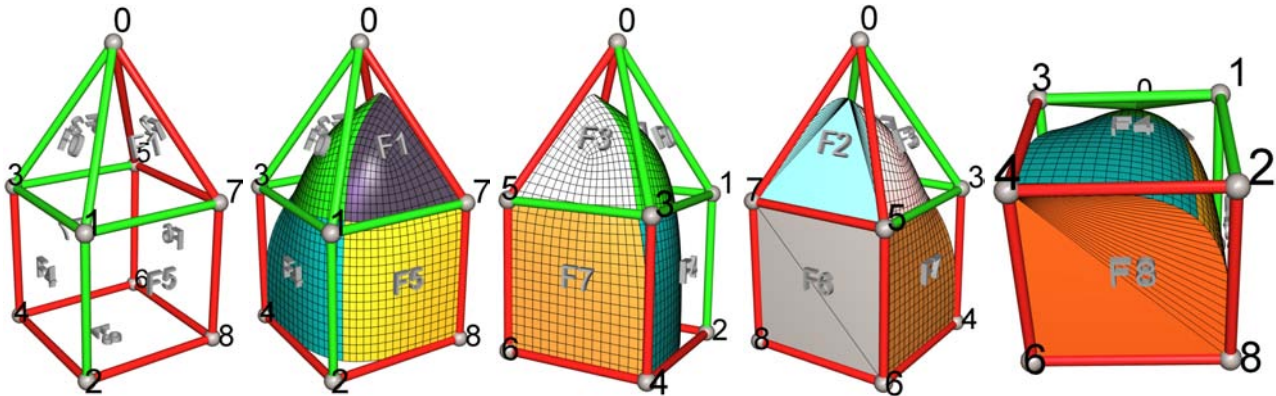


Figure 4.32: Combined B-rep classification example. The input mesh (a) is specified by the user. Mesh entities are classified and set up by `commitUpdate` and prepared for LOD assignment, tessellation, and rendering. Vertices and faces are classified according to tables 3.1 and 3.2. Vertex 1 is smooth, 3 is a dart, 0 and 2 are crease vertices, and 4, 5, 6, 7, 8 are corners. The multi-resolution faces, smooth faces 0, 1, 3, 4, 5, 7, and sharp faces 2 and 8, have resolution depth=4. Face 6 is a polygonal quad. Note that smooth faces can have corners, and that at least one neighbour of a sharp face must be a smooth face.

4.3.2 Combined B-Reps from a User’s Perspective: Manipulation, Update, and Rendering

This sub-section discusses how combined B-reps, or *cB-reps* for short, are used. The term ‘*user*’ means here ‘application programmer’, and not the end user of the application. The cB-rep data structure is listed in Fig. 4.33. It uses the trait mechanism to realize a patch complex by adding tessellation data as attributes to mesh entities. It is organized as (a) a mesh trait class, containing local classes V, E, and F for the entity attributes, and (b) the class `BRepCombined` that provides the API with all the methods for mesh manipulation and update, tessellation, and display. Although the entities carry a lot of information, there is a clear distinction between the *input data* that are specified by the user, and *output data* that store the tessellation computed by `BRepCombined::commitUpdate` (discussed in detail in the next section 4.3.3). As mentioned before, the amount of input is quite limited, essentially just two kinds of data:

- `BRepCombined::Vertex::position` is a 3D vector of type `Vec3f`,
- `BRepCombined::Edge::sharp` is the boolean flag for the edge sharpness (the same for both halfedges)
- existing entities may be changed, but they must be *touch*ed then.

According to a few rules, the user can arbitrarily create, change, and delete mesh entities as much as she wants. After a series of modifications, the tessellation of the surface needs to be set up, so that it is again in sync with the input data, and the object can be displayed. Technically, the function `commitUpdate` must be called, which takes care of the output data consistency for all entities.

A slight complication arises from the fact that the `Vertex`, `Edge`, and `Face` classes contain links to dynamically allocated data: chunks of points and triangles, patches, vertex and face rings. All dynamic data are stored in `skipchunks` and `skipvectors` (see sec. 4.2.1). This implies that the dynamic data must first be recovered, and put back to the pool for later re-use, when entities are manipulated or deleted. This leads to a few rules that must be respected when changing a mesh.

1. Allocation/deallocation methods.

Allocation of new entities is unproblematic. The `NewVertex` etc. methods are provided just for convenience, allocation can equally be done by directly accessing the mesh entity `skipvectors`. Note that combined B-reps use the `Edge::status` field to speed up the `mate` function (see `BRepMesh::newEdges` from Fig. 4.20).

Deallocation must be done with the `deleteVertex|Edge|Face` methods. They are very lean, since their only effect is to set the `deadID` field to the symbolic constant `DELETE` (halfedges: for the pair). Marked entities remain alive until `commitUpdate` is called.

2. Manipulations and touching.

Arbitrary changes can be applied to the connectivity. All `oneEdge`, vertex, face, and next pointers from Fig. 4.19 can be freely changed, as long as the mesh remains manifold, and the conventions from Fig. 4.21 are respected: no isolated entities, no `NULL` pointers, etc. All vertex positions and edge sharpness flags may also be changed at will. It is only mandatory to *touch* all changed entities, since otherwise changes are not detected and properly processed by `commitUpdate`. The following rules specify which entities need to be touched after changes.


```

struct BRepCombinedTrait
{
    struct V {
        typedef enum { CornerVertex,
                      CreaseVertex,
                      DartVertex,
                      SmoothVertex } Type;

        Vec3f position; // <=== INPUT DATA
        Type type;
        int ringID;
        // int status; // added by BRepMesh::Vertex
    };

    struct E {
        bool sharp; // <=== INPUT DATA
        int patchID;
        int sourceID; // pcB-rep ID
        // int status; // added by BRepMesh::Edge
    };

    struct F {
        typedef enum { HollowFace,
                      SmoothFace,
                      SharpFace,
                      PolygonalFace } Type;

        Type type;
        int ringID;
        int status;
        int depth; // <=== PER FRAME
        int depthSharp; // <=== PER FRAME

        int materialID;
        int triChunk;
        int sharpTriChunk[5];
        int sharpPtChunk;

        Vec3f normal;
        float normalDist;
        float normalCone;
        Vec3f sphereMid;
        float sphereRad;
    };
};

void render(BRepCombined& cbrep, float& quality,
            bool cbrep_was_changed)
{
    if (cbrep_was_changed) {
        cbrep.commitUpdate();
    }

    Vec3f eyepoint, zdir;
    float dx, dy;
    getViewConeGL (eyepoint, zdir, dx, dy);
    cbrep.framePrepare();
    cbrep.determineDepth(eyepoint, zdir, dx, dy, quality);
    cbrep.tessellateMesh();

    BRepCombined::Face* face;
    for (face = cbrep.faces.begin();
         face != cbrep.faces.end(); ++face) {
        if (face->active() && face->depth != -1) {
            cbrep.render(face);
        }
    }
    quality = cbrep.frameFinish(quality);
}

struct BRepCombined
{
    typedef BRepCombinedTrait Trait;
    typedef BRepMesh<Trait> Mesh;
    typedef Mesh::Vertex Vertex;
    typedef Mesh::Edge Edge;
    typedef Mesh::Face Face;
    typedef Trait::V V;
    typedef Trait::E E;
    typedef Trait::F F;

    // =====
    // 0. incremental allocation/deallocation

    Vertex* newVertex (const Vec3f& position);
    Edge* newEdges (bool sharp);
    Face* newFace ();
    void deleteVertex (Vertex* v);
    void deleteEdges (Edge* e);
    void deleteFace (Face* f);

    // =====
    // 1. mark changes

    bool touch (Vertex* v);
    bool touch (Edge* e);
    bool touch (Face* f);
    void touchIncident (Vertex* v);
    void touchIncident (Face* f);

    // =====
    // 2. process changes

    void commitUpdate ();

    // =====
    // 3. per frame: assign LOD

    void assignStaticDepth (int depth);
    void determineDepth (const Vec3f& eyepoint,
                        const Vec3f& zdir,
                        float dx, float dy,
                        float quality);

    void maxDepthSharpFaces ();

    // =====
    // 4. per frame: tessellate faces

    void tessellate ();
    void tessellate (Face* face);

    // =====
    // 5. per frame: render

    void framePrepare ();
    void render (Face* face);
    float frameFinish (float quality);

    Mesh mesh;
    TriangulatePolygon triangulator;
    TessellateCatmullClark tessellator;
    Skipchunk<ID> polygonalTriangles;
    Skipchunk<ID> sharpTriangles;
    Skipchunk<Vec3f> sharpPoints;
};

```

Figure 4.33: Combined B-rep trait.

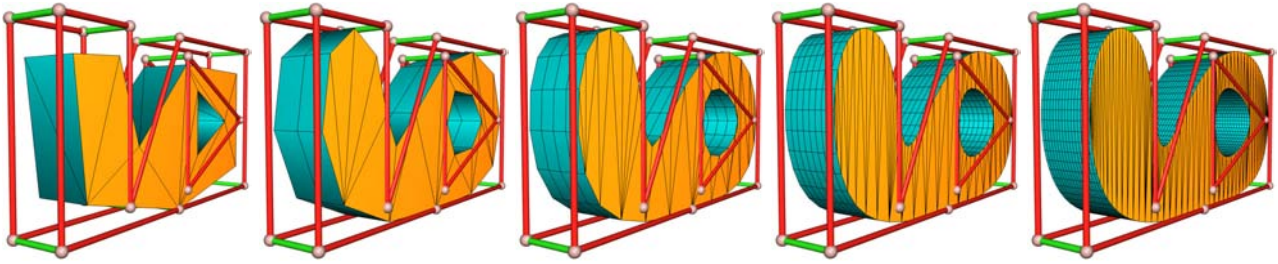


Figure 4.34: Multi-triangulation of sharp faces. Since smooth faces can be displayed in five different resolutions, sharp faces must have the same capability. The resolution of a sharp face equals the *maximum* resolution (depth) of its smooth neighbour faces. This is determined by `maxDepthSharp`. Note that neighbours with a lower resolution can adapt to this resolution: Smooth faces can always adapt to higher refined neighbours (see Fig. 3.37). Note that these triangulations are computed only on demand, i.e., all five `Face::sharpTriChunk` are set to `-1` at first.

- Basefaces and vertices where the connectivity changes must be touched. More precisely, a vertex must be touched when its edge ring changes. A face must be touched if its face boundary changes. In case of multiple boundaries, i.e., rings, the baseface must be notified in case any of the rings changes. In particular, entities that lose connections (ring removal) must also be touched!
- If the sharpness of an edge is changed (both halfedges), the edge must be touched. Note that touching an edge just means to touch the vertices and faces of the two halfedges.
- When a vertex is moved to a new position, **all incident faces must be touched**. This involves a loop, which is provided for convenience as `CombinedBRep::touchIncident(Vertex*)`. The reason is that the influence of a smooth vertex reaches out into its 2-neighbourhood, as it was illustrated in Fig. 3.17.

3. Render preparation: `commitUpdate`

After an arbitrary sequence of manipulations, `commitUpdate` must be called to prepare the rendering. Besides setting up the tessellation, it also (re-)computes the face normals, boundary spheres, planes, and normal cones.

Note that `commitUpdate` does not compute any tessellations. All the tessellations are computed only *on demand*. This means that, e.g., the triangulation of a sharp face, or a specific patch resolution, is not available until it is requested. When it is computed, it is cached though, to speed up the rendering in later frames.

4. Per Frame Step 1: LOD assignment.

The tessellation requests are formulated by setting the depth field of all basefaces, which may vary from frame to frame. For smooth faces, depth can be one of 0,1,2,3,4. This determines the number of subdivision steps, and thus the tessellation quality. For visible polygonal or sharp faces it is just 0. When a baseface (of any type) is not visible, because it is back-facing, or out of the view-frustum, its depth can be set to -1. After LOD assignment, the `maxDepthSharpFaces` function must be called, which determines the depth of all visible sharp faces (see Fig. 4.34).

For convenience, the `determineDepth` function is provided, which expects the current *view cone*, and assigns a LOD to all faces according to an overall quality, usually between 0 and 1. The view cone is defined by the current eye position, a normalized view direction, and an opening angle (dx,dy): dx is the ‘min-LOD’ distance, usually the far plane distance, and dy is the maximum distance of a visible point on the min-LOD plane from the view axis. `determineDepth` also executes `maxDepthSharpFaces`.

5. Per Frame Step 2: Tessellation.

It may be that for some faces, the requested LOD is not available from the cache, and must be computed. The `tessellate` routine loops over all faces to assure all triangulations and subdivisions exist in the respective caches.

LOD assignment and tessellation are not integrated with the actual render routine to permit the use of multi-threading, so that the tessellation is computed in a separate worker-thread, possibly running on a second CPU.

6. Per Frame Step 3: Rendering.

This involves just to call `render` for all visible faces. This can be done in any order, in particular, faces can be grouped by material, as to reduce OpenGL material state switches. This can have a great impact on performance.

The `framePrepare` and `frameFinish` routines are provided for convenience. They measure the time it took to render, and adjust the overall quality, as to keep the frame rate always just above 20 fps.

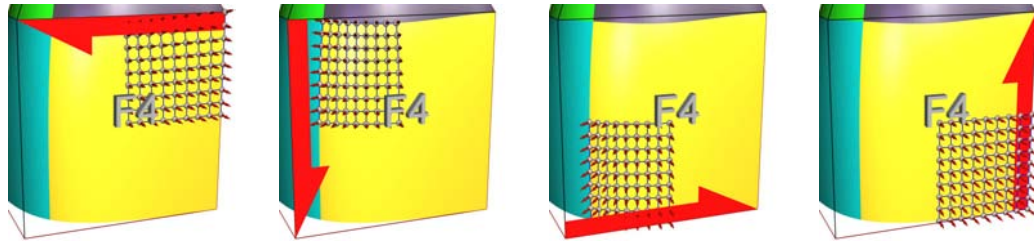


Figure 4.35: One Catmull/Clark patch of 9×9 vertices and normals belongs to every halfedge of a smooth face. Its ID is stored in the `patchID` field of the halfedge. It is -1 if the halfedge's face is not smooth. Note: It is the patch at the halfedge's *source* vertex which belongs to a halfedge. This is consistent with the rule that `edge->vertex` is the source vertex of a half-edge. This is also the vertex point of the patch, whereas the face midpoint is the face point of the patch.

4.3.3 The Combined B-Rep Data Structure

The previous section has given an outline of how to use the combined B-Rep data structure. This section looks 'under the hood' and discusses the data items of the `CombinedBRep` from Fig. 4.33 in more detail. This is important, e.g., for accessing the different bits of information stored at the different locations. Note that in the actual implementation, space can be saved by combining different data items with limited ranges (e.g., `F::depth`) into one bit-field, and also by alternative uses: A sharp face has no `ringID`, and a smooth face has no `sharpPtChunk` (explained below).

The data in `BRepCombined::Vertex`. According to table 3.1, the vertex type is essentially the number of sharp edges incident to a vertex. An example of a classification is shown Fig. 4.32. A vertex of any type may be incident to a smooth face, in which case `ringID` is the index of the vertex ring, stored with `TessellateCatmullClark`. For corners without any incident smooth faces, `ringID` is -1 .

Due to the wedge problem (sec. 4.1.4), a cB-Rep vertex contains no vertex normal: The normal is only defined with respect to a face. For an incident smooth face, the vertex normal, and also the 'true' position of the vertex on the surface, can be accessed via a vertex ring, explained in the next section 4.3.4. Also see sec. 3.4.1 for background information on vertex rings. For polygonal or sharp faces (with corner or crease vertices), the vertex normal is just the face normal, since such faces are assumed to be planar. The vertex normal with respect to a smooth face can also be directly accessed, with a method that is presented in section 4.3.4.

The status is set to the symbolic value `NEW=0` for new vertices, `READY` for active vertices, and `DELETE` if the vertex is marked for deletion. It is eventually removed the next time when `commitUpdate` is called, which performs the actual deactivation. When the vertex needs just an update, the status is `TOUCHED`.

The data in `BRepCombined::Edge`. In case the edge's face is smooth, `patchID` is the index of a Catmull/Clark patch, and -1 otherwise, see Fig. 4.35 for the example of a degree 4 face. Patches are maintained by the `TessellateCatmullClark` class from the next section. To speed up the `mate` method, the `Edge::status` is 0 for the first, and 1 for the second halfedge of a halfedge pair (see no. 5 in Fig. 4.21). The `sourceID` is for the progressive version of combined B-reps (section 4.4).

The data in `BRepCombined::Face`. The type is the face classification (*smooth*, *sharp*, *polygonal*, *hollow*) from table 3.2, also see Fig. 4.32. When the face is smooth, `ringID` is the face ring index from `TessellateCatmullClark`, -1 otherwise. When the face is sharp, `sharpTriChunk` and `sharpPtChunk` store a multi-triangulation for the face (see Fig. 4.34).

The depth field is the LOD of smooth faces, 0 for visible polygonal faces, and determined automatically for visible sharp faces by `maxDepthSharpFaces`. It is set to -1 for faces that are not to be shown. Polygonal faces provide only a single resolution, and they store only a single chunk of triangles, whose ID resides in `triChunk`. But note that *all* faces have a `triChunk`, and may thus be rendered as polygonal faces. That makes it possible to switch back to polygonal rendering altogether, which can sometimes be quite informative.

The normal and `normalDist` store the (approximate) face plane equation. The `normalCone` is the sine of the opening angle of the face normal cone, determined by the maximum deviation of the tessellation normals from the face normal. The `sphereMid` is the face midpoint for polygonal and sharp faces, and the face point for smooth faces. The `sphereRad` is the maximum distance of any point from the tessellation from the midpoint `sphereMid`.

The status is `NEW=0` when a face is new, `READY` for active faces, and `DELETE` if the face has been marked for deactivation, i.e., to be removed by `commitUpdate`. When the face has been manipulated, the status is `TOUCHED`. The `materialID` establishes the link to some given material library.

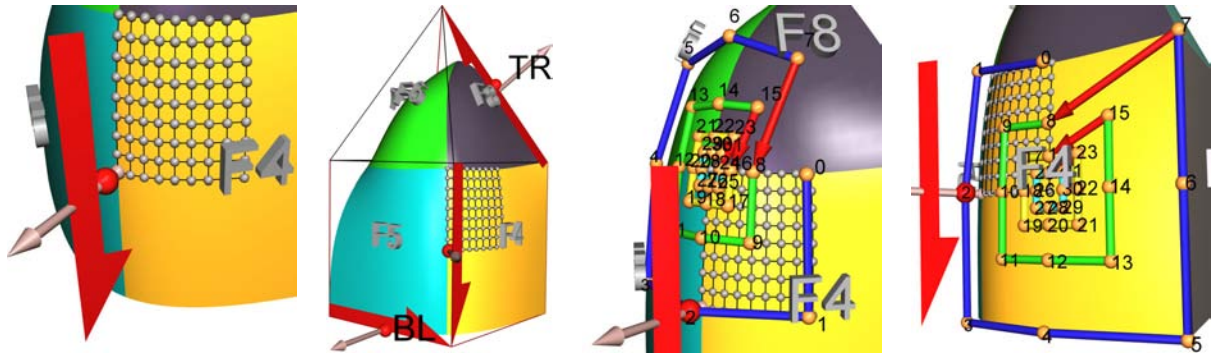


Figure 4.36: Illustration of the data in a patch structure. Every halfedge has a patch attached to its source vertex.

The data in BRepCombined. Vertices, edges, and faces have only handles to dynamic tessellation data. The location where these data are actually stored are the five data members of BRepCombined:

- **triangulator** is one instance of the polygon triangulation algorithm from sec. 4.2.2 (code in Fig. 4.24). Since it generates (and contains) dynamic data, it is more efficient to keep one instance alive, than to create and destroy a triangulator every time one is needed. The triangulator is used both for polygonal and sharp faces.
- **polygonalTriangles** contains the triangulation of polygonal faces. There is one chunk for each face, produced, e.g., as in `Trait_VpFntc::triangulate()` from Fig. 4.24, so that a polygonal face can be rendered just as in Fig. 4.25: The IDs in the triangulation are the vertex IDs, e.g., they directly refer to `mesh.vertices`.
- **sharpPoints** and **sharpTriangles** contain the triangulation of sharp faces. The vertices of sharp faces are not mesh vertices, since part of the boundary is a B-spline curve. So the boundary is sampled at the highest resolution potentially needed to match a smooth neighbour, at 16 points per B-rep edge. The sampling is stored in the face's `sharpPtChunk`, which refers to the `sharpPoints` skipchunk. Each face can have up to five triangulations, corresponding to boundary resolutions of 1, 2, 4, 8, or 16 line segments per mesh edge (see Fig. 4.34). For each of these triangulations, there is one chunk of triangles in `sharpTriangles`. The indices of these triangulations all refer to the same boundary `sharpPtChunk`.
- **tessellator** is a separate data structure that stores vertex rings, face rings, and patches (see next section 4.3.4).

Sharp faces contain indeed five different triangulations since, in general, coarser triangulations cannot be simply obtained by straightforward downsampling of triangulations from a higher level. This is especially the case for non-convex faces, as can be seen not only in Fig. 4.34, but also in Fig. 4.23, which actually shows three different triangulations of a sharp face. Although this looks like a large overhead, it requires only essentially twice the space of the highest resolution: The triangulation of a simple polygon with n vertices has $n - 2$ triangles, so the triangulation of a sharp face with n crease vertices has $16n - 2$ triangles, which is more than the $(8n - 2) + (4n - 2) + (2n - 2) + (n - 2) = 15n - 8$ triangles for the other four triangulations. The ratio is worse, of course, when a face has many corner vertices on its boundary, so that it contains straight line segments that are no B-spline curves. The distinction between BRepCombined and BRepCombinedTrait is for purely technical reasons: BRepCombined has member functions such as `render(Face*)`, but `Face*` is only properly defined *after* the definition of the trait, and not *within* the trait.

4.3.4 The Data Structure for the Tessellation of Smooth Faces

The tessellation of smooth faces of the mesh is stored in a stand-alone data structure, the class `TessellateCatmullClark`, shown in Fig. 4.37. This class does not know anything about meshes. It replicates all relevant information from the mesh, so that it can do without any pointers to mesh entities. The advantage is that the somewhat complicated subdivision computations are localized in memory, which greatly improves performance, because CPU cache misses are minimized. The price to pay is a temporary memory overhead: In principle, most of the replicated data can be thrown away as soon as the patch computation is completed, i.e., the patch grids are entirely filled with valid points.

The `TessellateCatmullClark` class defines two local classes `Ring` and `Patch`. Every smooth face of the mesh, and every vertex that is incident to a smooth face, contains one `ringID` index, which refers to a skipvector of rings, `TessellateCatmullClark::rings`. Every halfedge of the mesh that belongs to a smooth face contains one `patchID` index, referring to the skipvector `TessellateCatmullClark::patches`. Rings and patches, in turn, also directly refer to each other via indices.

Besides these skipvectors, and a skipchunk for the ring ring points, `TessellateCatmullClark` contains a table of the various subdivision weights (from sections 3.1.4, 3.1.5, and 3.1.6) for each valence, which can also be extended on demand. Finally, it contains the pre-computed triangle strip indices for the configurations from section 3.4.3.


```

struct Ring {
    int type;

    Vec3f point[4];
    Vec3f limitPoint, limitNormal;
    int chunkID;
    int nextring;
    int valence;
    int depthMax;

    Vec3f& point () { return point[0]; }

    int& deadID ();
    bool active () const;
};

struct Patch {
    Vec3f edgepoint;
    Vec3f limitEdgePoint
    Vec3f limitEdgeNormal;

    bool sharpLeft, sharpTop;
    int vertexRing, vertexIndex;
    int faceRing, faceIndex;
    int patchBL, patchTR;
    int depthMax;
    int status;

    int depth, depthLeft, depthTop;

    float vertex[81*3];
    float normal[81*3];

    int& deadID ();
    bool active () const;
};

struct TesselateCatmullClark
{
    TesselateCatmullClark () { }
    ~TesselateCatmullClark () { }

    struct Ring { ... see code to the left };
    struct Patch { ... see code to the left };

    // 1. setup sharpness & face points =====
    void faceRingBegin (int facering);
    void faceRingEdge (bool isSharp, int patchID,
                       const Vec3f& vertexpos);
    void faceRingEnd ();

    // 2. setup vertex rings =====
    void vertexBegin (int vertexring, int type,
                     const Vec3f& vertexpos);
    void vertexEdge (int patch0, int patch1,
                     const Vec3f& matepos);
    void vertexEnd ();
    void setTR (int dest, int src);
    void setBL (int dest, int src);

    // 3. setup patch connectivity =====
    void faceBegin (int facering);
    void faceEdge (int vertexring, int patchID);
    void faceEnd ();

    // 4. initialize changed patches =====
    void setupPatches ();

    // 5. tessellate =====
    void tessellate (int patchID, int depth,
                    int depthLeft, int depthTop);

    // 6. render =====
    void render (int patchID);

    void setupVertexRing ();
    void refineRing (Ring* ring);
    void setupPatch (Patch* patch);
    void computeWeights (int n);

    Skipvector<Patch> patches;
    Skipvector<Ring> rings;
    Skipchunk<Vec3f> ringpoints;
    Skipchunk<float> weightTable;
    vector<ID> stripIndices;
};
    
```

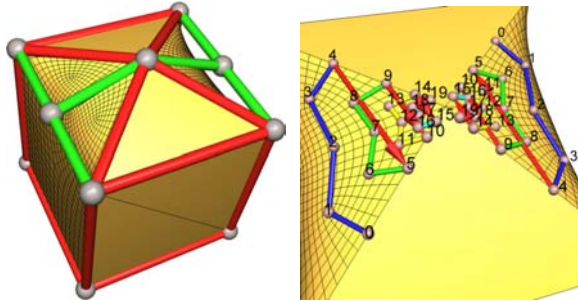


Figure 4.37: The Catmull/Clark tessellation class with its local classes Ring and Patch.

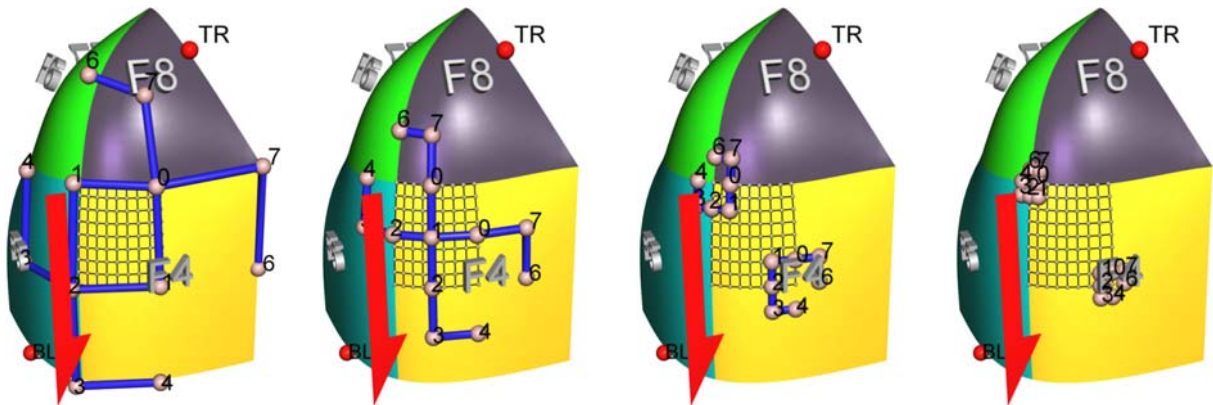


Figure 4.38: The patch control mesh is made up of seven CVs from the vertex and face rings on each level.

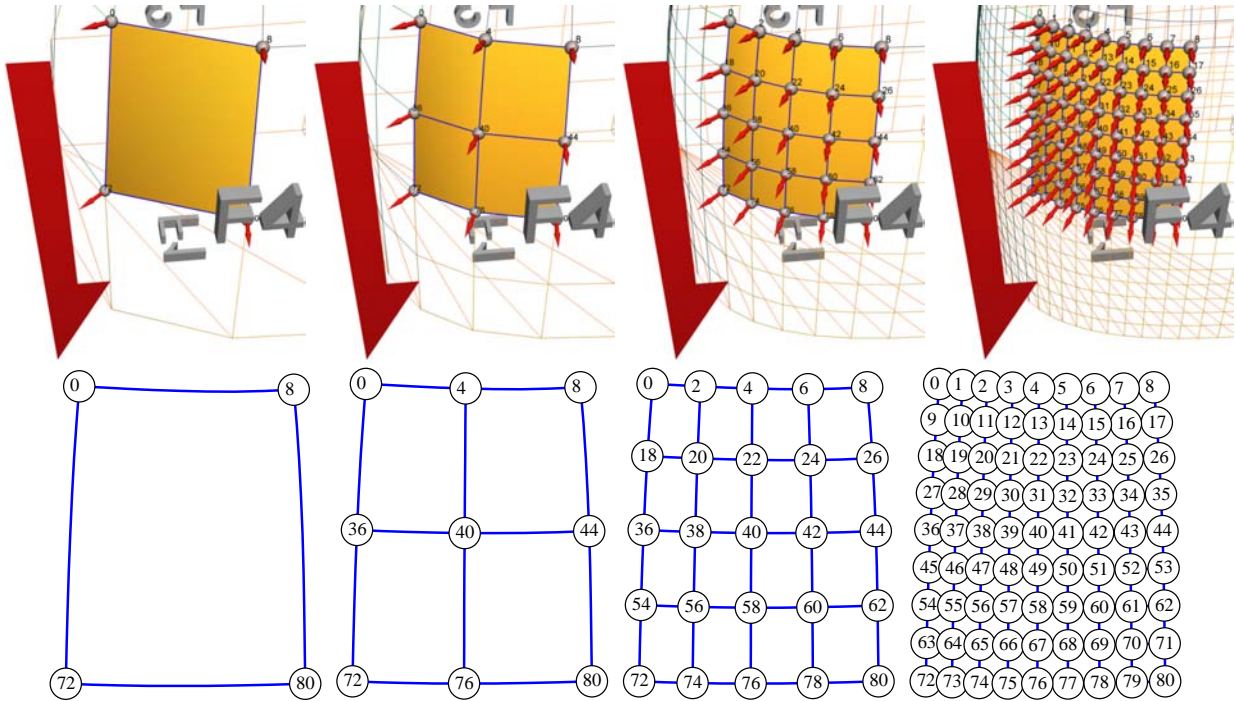


Figure 4.39: Different uniform patch resolutions, and the corresponding point enumerations.

Patch data. The relation between rings and patches is detailed in Figs. 4.36 and 4.38. Fig. 4.36 (a) first shows the essential components of a halfedge’s patch: The grid of 81 points (and normals, not shown here), stored in the float arrays `Patch::vertex` and `Patch::normal`, the edge point `Patch::edgepoint` from the first subdivision, and the limit edge point with its normal, `Patch::limitEdgePoint` and `Patch::limitEdgeNormal`. The latter two are replicated from the grid for faster access.

The orientation of the patch in Fig. 4.36, with the halfedge pointing downwards, is no coincidence: The four boundaries of the patch are denoted the top, bottom, left, and right boundaries, or T, B, L, and R for short. This orientation also corresponds to the row-wise enumeration of the points in the grid (see Fig. 4.39, which also shows the vertex normals):

- the TL grid vertex has index 0, and is the vertex limit point
- the TR grid vertex has index 8, and is the edge limit point of the CCW previous patch
- the BL grid vertex has index 72, and is the edge limit point of the patch
- the BR grid vertex has index 80, and is the face limit point.

The right and bottom boundaries lie in the face interior, so they are always smooth. Only the top or left boundaries may be sharp, which affects the choice of the tessellation scheme to be used. The patch therefore stores the respective sharpnesses as `Patch::sharpLeft` and `Patch::sharpTop`, which makes for the four different possible combinations.

In the standard case, where both edges are smooth, the control mesh of the patch is given by the vertex and face rings, and two supplemental points in the top right and bottom left corners. Since the edges are smooth, the respective neighbour faces are also smooth, and the CVs are in fact edge points, as illustrated in Fig. 4.36 (b):

- the TR CV is the `Patch::edgepoint` from `edge`→`faceCW()`→`mate()`→`faceCCW()`
- the BL CV is the `Patch::edgepoint` from `edge`→`mate()`→`faceCW()`

Since there is no connection to the mesh, the patch actually stores the indices of the respective patches containing these points, in `Patch::patchBL` and `Patch::patchTR`.

Each patch is incident to one vertex ring and one face ring, which capture the potential irregularity of the vertex and face (explained in detail in section 3.4.1 from chapter 3), whose indices are `Patch::vertexRing` and `Patch::faceRing`. But only seven CVs from the rings on each level are actually relevant for the patch. The relative position of the patch with respect to the rings is given by `Patch::vertexIndex` and `Patch::faceIndex`. These two indices tell the index of the `Patch::edgepoint` in both of the rings. In Fig. 4.36 (c) and (d), this happens to be 2 for the vertex ring, and 2 for the face ring (again note the CW orientation of the vertex ring and the CCW orientation of the face ring).

From these informations, the control mesh of the patch on the different levels can be readily obtained. Fig. 4.38 (a) shows the patch control mesh from the first subdivision, in this example made of the CVs (6, 7, 0, 1, 2, 3, 4) from both the vertex ring and the face ring, and the TR and BL CVs.

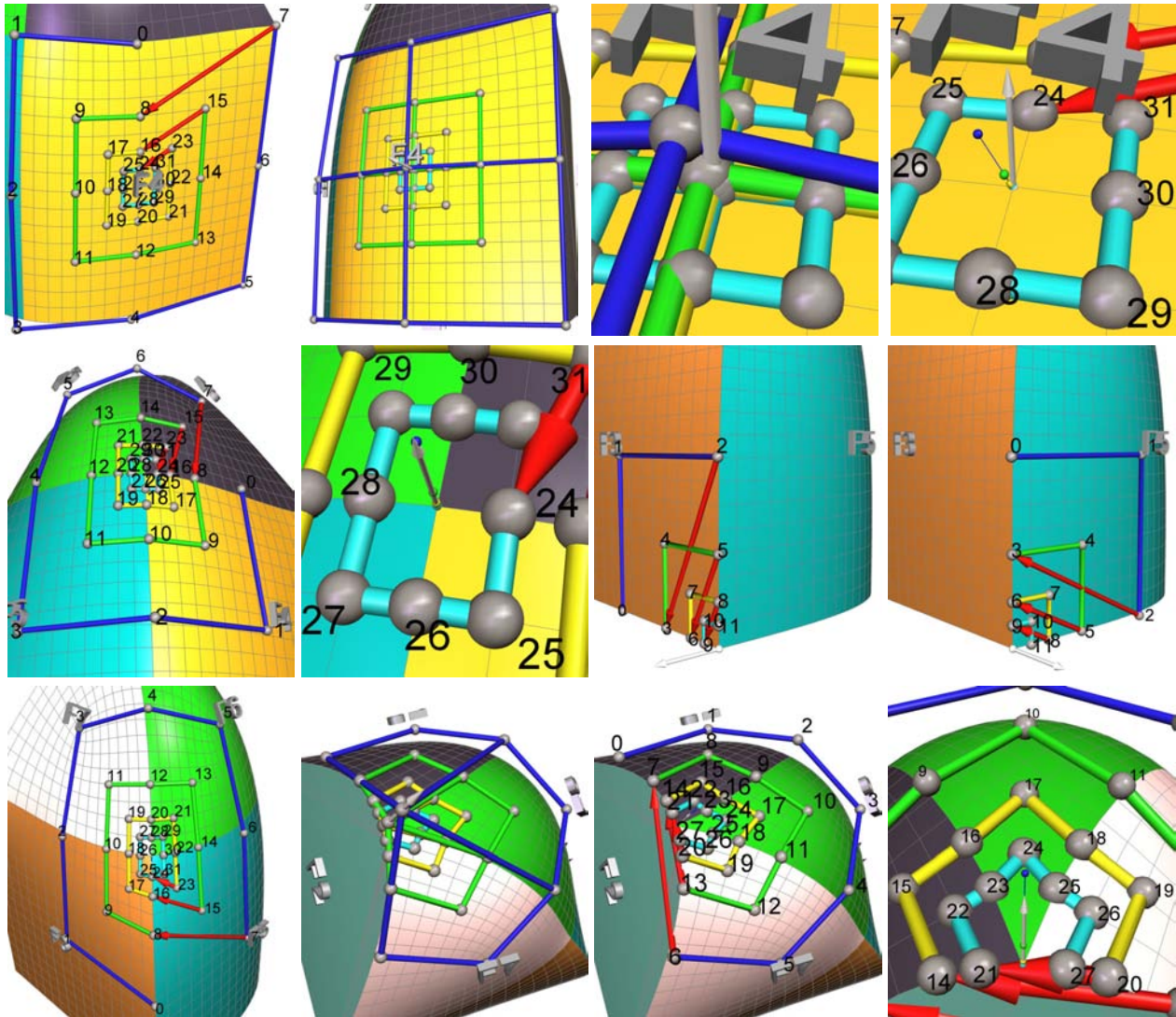


Figure 4.40: Top row (1a-d): Illustration of face ring data. Lower rows (2a-d) and (3a-d): Different cases of vertex rings. (2a, b): Smooth vertex with no sharp edges. (2c, d): Corner vertex with more than two sharp edges. (3a): dart vertex with one sharp edge. (3b, c, d): Crease vertex with two sharp edges.

Ring data. The same data structure is used for vertex and face rings, namely `TessellateCatmullClark::Ring`, illustrated in Fig. 4.40 (1a-d) for the case of a face ring. The valence of this face is 4, which is stored in `Ring::valence`. The number of CVs in such a closed ring is $2 \cdot \text{valence}$. But a ring must be available on all subdivision levels 1-4, so in fact $4 \cdot 2 \cdot \text{valence}$ ring points are stored. They are contained in one consecutive array, always beginning with an (arbitrarily chosen) edge point. The canonical enumeration order for the 32 points in the face ring of a valence 4 face is illustrated in Fig. 4.40 (1a). Since the ring point array is of dynamic size, it is stored in a skipchunk, whose ID is stored in `Ring::chunkID`, referring to `TessellateCatmullClark::ringpoints`.

The face points for all levels are stored in the array `Ring::point[4]`. Point 0 in it, or `Ring::point()`, is the face point from the first subdivision. Together with the rings on the various levels they define the ring of quadrangles around the face center, shown in Fig. 4.40 (1b). The situation in the center is zoomed in closer to in Figs. 4.40 (1c) and (1d): The face point, shown as blue, green, yellow, and cyan balls in (1d), rapidly converges to the limit position and normal (grey), which is stored in `Ring::limitPoint` and `Ring::limitNormal`.

The situation for vertex rings can be a little more complicated. Vertices can be incident to any sequence of smooth and sharp edges. The number of sharp edges determines the vertex type, and is stored in `Ring::type`, which can be 0,1,2, or 3. The different cases are shown in the lower two rows of Fig. 4.40. The organization of smooth vertex rings (4.40, 2a) is identical to the one of face rings, except for the CW orientation. A vertex ring also starts with an edge point, and there is a unique limit vertex position (4.40, 2b). For dart vertices, the vertex ring starts with the edge point of the sharp edge

(4.40, 3a). This principle also applies to the case of several sharp edges: The vertex ring of a crease vertex both starts and ends with a sharp edge. In this case, the `Ring::valence` is *not* the same as the vertex valence. Instead, the ring valence is the valence of the smooth wedge, i.e., the number of consecutive smooth faces. Fig. 4.40 (3b, c, d) show a crease vertex ring with valence 3, where each of the four levels contains $(2 \cdot \text{valence} + 1)$ points. The enumeration scheme is analogous to the previous examples.

Corner vertices finally can also exhibit alternating configurations of polygonal and smooth wedges. Fig. 4.37 shows a valence 6 corner vertex with two smooth wedges of valence 2 each. In case of more than one smooth wedge (which may also happen with crease vertices), the `Ring::nextRing` field permits to access all vertex rings attached to one mesh vertex (via the `Vertex::ring` field, of course). Yet from a patch, the respective vertex ring can always be directly accessed with `Patch::vertexRing`. Of course, there does not have to be a polygonal wedge between smooth wedges; and a smooth wedge may very well consist of only a single face. This is illustrated in Fig. 4.40 (2c) and (2d).

Vertex normals. The described mechanism also eventually yields a method to access the vertex normal with respect to smooth faces. With every smooth wedge, a mesh vertex has a unique normal vector. So the vertex normal is well defined only with respect to a particular face incident to the vertex. A face, however, can be selected by choosing one outgoing halfedge: Recall that the `edge→vertex` is the halfedge's source vertex. So given a halfedge, the vertex normal is determined as follows:

- If the face is polygonal or sharp, or the vertex belongs to a ring of a polygonal or sharp face, the vertex normal is the face normal: `edge→face→normal`.
- If the face is smooth, the vertex normal is the limit normal of the vertex ring of the halfedge's patch: `rings[patches[edge→patchID].vertexRing].limitNormal`

4.3.5 CommitUpdate, Tessellation, and Rendering

The previous sections described the various data items in a valid combined B-rep. This section shows now how the output data are generated in the first place, and how they are used for depth assignment, tessellation, and rendering.

The `commitUpdate` routine. This routine is called when a new mesh with valid input parameters (vertex positions and edge sharpness) is created, and whenever a valid combined B-rep has been manipulated and all affected entities were touched. The name `commitUpdate` is motivated by a useful analogy to database interfaces, where sequences of atomic update operations are temporarily stored, until they are finally *committed* to actually change the contents of the database. The signal that commitment is necessary for combined B-rep mesh entities is realized by the status field. Recall that vertices and faces have a status field that can have either of the symbolic values NEW, READY, TOUCHED, or DELETE. The routine has to proceed in six stages, which are summarized in the following, to make all active entities READY again.

1. Garbage collection for triangulation data

Skipchunks use one large array that contains all chunks, each of them a coherent array of small data items, usually IDs or floats. Deletion of a chunk only switches off one sub-array, new chunks are always allocated at the end. The large array is purged only when the number of deleted items becomes too large, which is for instance the case when `range() > 5·size()`. This is an issue for the triangulation data for polygonal and sharp faces: `polygonalTriangles`, `sharpTriangles`, and `sharpPoints` from `BRepCombined`. Unused items are removed in a garbage collection fashion, all three skipchunks are purged if the criterion applies to either of them. Purging implies that the Face chunk IDs are updated.

2. Propagate TOUCH from faces to vertices

This involves an iteration over all halfedges. Touching is face-centered: When a face is touched, all neighbouring faces may become subject to re-classification, due to the range of subdivision (see Fig. 3.17 from section 3.2.1). This is realized by touching a halfedge's vertex whenever its (base-) face is touched. At the same time, halfedge pairs marked for deletion are deactivated (in inverse order, for later re-use), together with the patches they have.

3. Classify touched vertices, propagate TOUCH from vertices to faces

This involves an iteration over all vertices. New and touched vertices must be classified before their faces can be classified. With one `vertexCW` loop around each such vertex, the incident sharp edges are counted to determine the vertex type: smooth, dart, crease, or corner. In case the vertex has vertex rings (i.e., it was previously incident to one or more smooth wedges), but the number of smooth edges found is zero, the vertex ring has to be precautionarily deleted: In this case it may be that the vertex is surrounded exclusively by polygonal faces. – In the same loop, the vertices marked for deletion are deactivated, together with their vertex rings, if they have any.

4. Classify touched faces, prepare face rings for smooth faces

This involves an iteration over all faces. The first thing to do for new or touched faces is to remove the (sharp and/or polygonal) triangulation, if there is one. Then the face type is determined as follows:

- If the face is a ring, its type is `HollowFace`.
- If the face has rings, its type is `SharpFace` in case it has a crease vertex, and `PolygonalFace` otherwise. This classification is irrespective of whether the face has any smooth edges: Smooth faces cannot have rings.
- The face is classified as `SmoothFace` only if it has a smooth edge where the respective neighbour face does *not* have rings. Otherwise, this edge is not counted as a smooth edge.
- As a last resort, the face is classified as sharp or polygonal, depending on whether it has a crease vertex.

If the face is not smooth, but has a face ring, the face ring is deactivated. If the face is a sharp face, the boundary polygon is sampled (with 16 line segments per B-spline edge), creating the face's `sharpPtChunk`. For all sharp and polygonal faces, the (approximate) face normal vector with respect to the face's B-rep vertices is computed, as well as the (approximate) bounding sphere. Its midpoint lies in the face plane, which is determined as well.

If the face is smooth, one cyclic loop along the face boundary provides all halfedges with patches. At the same time, the `faceRingBegin`, `faceRingEdge`, and finally `faceRingEnd` routines from `TessellateCatmullClark` set the sharpness flags for the patches, set the face ring valence, and compute the face point of the 1st subdivision as the centroid of the vertex positions. Note that the face ring can *not* be collected already at this stage, since the edge points have not yet been computed.

All the smooth faces, as well as all of their vertices, are temporarily tagged with a new status `SMOOTHFACE`. All other vertices and sharp or polygonal faces receive the status `READY`. All their triangulation chunk IDs are set to `-1`: Triangulations are computed only on demand.

5. Set up vertex rings for vertices of smooth faces

This involves a loop over all vertices tagged as `SMOOTHFACE` vertices. The number of smooth wedges is determined, and for each vertex at least one vertex ring is set up, by one `vertexCW` loop starting from the first sharp edge found (if there is any). This involves calling the functions `vertexBegin` once, then `vertexEdge` for each edge, and finally `vertexEnd`, from `TessellateCatmullClark`. These functions take care of properly beginning and finishing one vertex ring for each smooth wedge, i.e., for each sequence of consecutive smooth faces. In the same `vertexCW` loop, the indices of the BL and TR patches, as well as the `vertexRing` and `vertexIndex`, are set for each visited patch. These visited patches are just the patches for which the control mesh may have changed (compare with Fig. 4.38). They are labeled as `TOUCHED` using the `Patch::status` field. Note that a touched patch does not have to belong to a face that was touched.

Also note that `vertexEdge` has to do the same as `vertexEnd`, namely call `setupVertexRing`, whenever a smooth wedge is finished. This is the case when the wedge's closing sharp edge arrives, which can be told from the sharpness flags stored in the patches: For each outgoing halfedge, `vertexEdge` is called with the mate vertex position, and the `patchID` indices of both the edge and its mate (which may also be `-1`, in case of incident non-smooth faces). This provides enough information to also compute the edge points (from the 1st subdivision), because all face points computed in stage 4 now properly exist. The edge points are stored in the patches, and edge points and face centroids are copied to the first level vertex ring. Whenever a smooth wedge is finished, the ring valence is set, space for higher level rings is allocated, and the vertex limit point and normal are readily computed. Finally, the vertex is marked as `READY`.

6. Set up face rings, normal cones, and bounding spheres of smooth faces

This involves a loop over all faces tagged as `SMOOTHFACE`. One pass over the face boundary eventually yields the face ring, the alternating sequence of edge and vertex points from the 1st subdivision, as well as the limit point and normal of the face point. And it is only now that the `faceRing` and `faceIndex` can be stored in the face's patches.

The (approximate) bounding sphere of a smooth face is obtained via an axis-aligned bounding box containing all vertices as well as the limit vertex positions, and the face limit point; the center of the box becomes the center of the bounding sphere. The view cone is the sine of the maximum angle between the face limit normal and a vertex limit normal. This completes the face setup, and the face status can now be set to `READY`.

Finally, all patches with status `TOUCHED` are initialized: The four grid corners (with indices 0, 8, 72, and 80) are initialized from the vertex, face, and edge limit points and normals, and the patch status becomes `READY` as well.

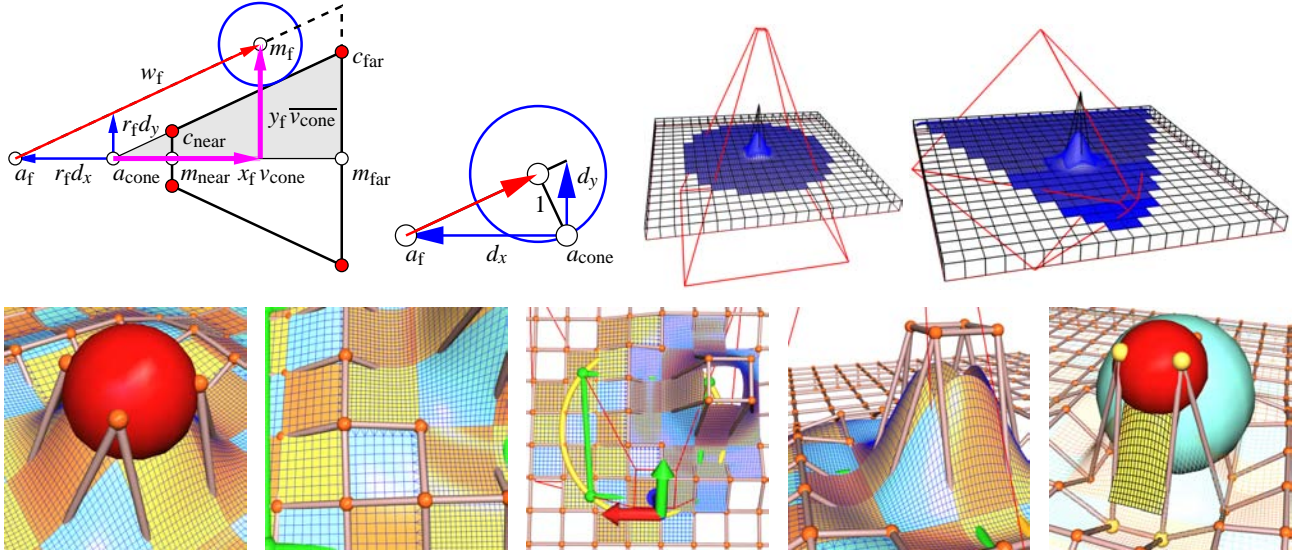


Figure 4.41: Apex translation of the view cone. (1a): The apex is translated in negative view direction to reduce the intersection test between face bounding sphere and view cone to a point-in-cone test. (1b): Lengths d_x and d_y are determined once for the unit sphere. (1c): The view cone has a slight excess with respect to the view frustum, but culling is much faster to compute. (1d): The intersection of the view cone and a plane is a conic section.

Bottom, (2a): The bounding sphere of a smooth face contains the vertices, their limit points, and the face limit point. It is more efficient to compute than the exact bounding sphere of the tessellation, and appropriate for culling: (2b) shows a view, and (2c) shows the view frustum and cone (ellipse) from above. (2d): The peak of the hill is correctly clipped away, since the corner ray of the view frustum does not hit it. (2e): A problem with bounding spheres is that a long, thin face (yellow, with yellow vertices) leads to an overly large bounding sphere (cyan).

View frustum culling using the view cone. In each and every frame, the first thing to do is to find out which faces are visible. All mesh faces have face normals and bounding spheres. Smooth faces additionally provide a normal cone, which measures the curvature of the patches that make up the face. These bits of information can be used to quickly identify faces that lie outside of the view frustum, and faces that are back-facing: The great advantage of combined B-reps, as a realization of the ‘patch complex’ idea from section 2.4, is that their faces usually comprise a greater number of triangles. This makes culling worthwhile.

Frustum culling is done for each face via *apex translation* of the *view cone*, as shown in Fig. 4.41. The use of a view cone instead of a view frustum is based on the assumption that the aspect ratio of the viewport is usually close to 1. The cone is determined only once per frame. It is defined by an apex a_{cone} , which is the eye position, the axis v_{cone} , and the slope s_{cone} (instead of the opening angle). The axis goes through the midpoints m_{near} and m_{far} of near- and far-plane (in case of a symmetric view frustum). The apex is the intersection of the line through two frustum corners c_{near} , c_{far} and the cone axis. Using OpenGL, the world coordinates of frustum corners can be obtained using `gluUnProject`. The slope is the width increment per depth, i.e., $s_{\text{cone}} = |m_{\text{far}} - c_{\text{far}}| / |m_{\text{far}} - a_{\text{cone}}|$.

The crucial observation is that the overlap test between the view cone and a bounding sphere (m_f, r_f) of a given face f can be reduced to a point-in-cone test, simply by using a translated apex $a_f = a_{\text{cone}} - (r_f d_x) v_{\text{cone}}$. The ‘unit displacements’ d_x and d_y are the displacements that would be needed for apex translation in case of a unit sphere. They are determined only once (per frame) by solving two equations. The first equation is $d_y/d_x = s_{\text{cone}} \Leftrightarrow d_y = d_x s_{\text{cone}}$. The second equation states that the height of the right triangle with catheta d_x and d_y is 1. Using similar triangles (Fig. 4.41, 1b), this gives:

$$\frac{d_y}{1} = \frac{d_x}{\sqrt{d_x^2 - 1}} \Leftrightarrow (d_x^2 - 1) d_y^2 = d_x^2 \Leftrightarrow (d_x^2 - 1) d_x^2 s_{\text{cone}}^2 = d_x^2 \Leftrightarrow d_x^2 = \frac{1}{s_{\text{cone}}^2} + 1$$

Since the view vector v_{cone} is normalized, no square root is necessary for testing whether the sphere center m_f is inside the cone with modified apex a_f : Let $w_f := m_f - a_f$ (red vector in Fig. 4.41 (a)), be the vector from a_f to the midpoint m_f . It can be written as the decomposition $w_f = (x_f + r_f d_x) \cdot v_{\text{cone}} + y_f \cdot \overline{v_{\text{cone}}}$ into orthogonal components v_{cone} and $\overline{v_{\text{cone}}}$ with $\langle v_{\text{cone}}, \overline{v_{\text{cone}}} \rangle = 0$ and $|v_{\text{cone}}| = |\overline{v_{\text{cone}}}| = 1$. The distance from the apex to the foot-point from m_f onto the view axis is $x_f = \langle w_f, v_{\text{cone}} \rangle$, and $r_f d_x$ is the distance from the apex to the translated apex. The component orthogonal to the axis is $\overline{w}_f := w_f - x_f \cdot v_{\text{cone}} = y_f \cdot \overline{v_{\text{cone}}}$. So the squared length of this vector is $y_f^2 = \langle \overline{w}_f, \overline{w}_f \rangle$. The midpoint is inside the translated cone if and only if $y_f^2 / (x_f + r_f d_x)^2 < s_{\text{cone}}^2$. – So view cone culling is extremely fast to compute.

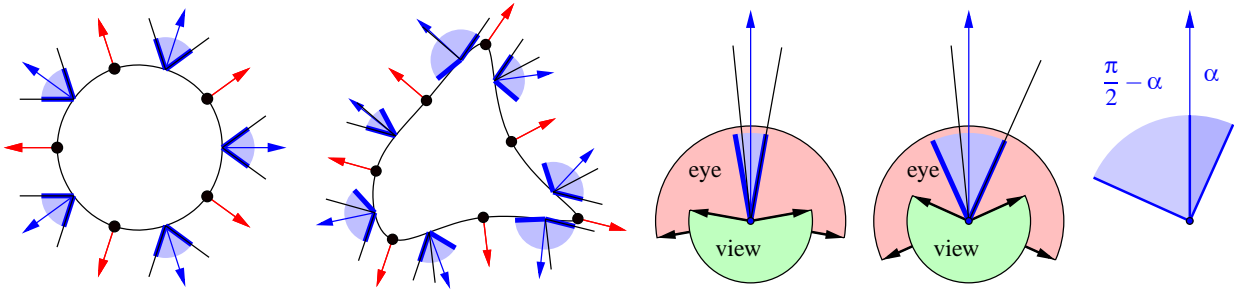


Figure 4.42: Back-patch culling using the normal cone. For every smooth face, the maximum deviation of a vertex normal (red) from the main face normal (blue) is measured (a),(b). This yields the (blue) normal cone with opening angle 2α . (c,d): The face is visible if the eye point lies the ‘eye’ section, or, equivalently, if the negative eye direction lies in ‘view’. (e): The angle between view direction and face normal must be larger than $\frac{\pi}{2} - \alpha$.

Back-patch culling. A face that passes the view cone test may still not be visible because it is back-facing. At least for closed manifold surfaces, the backside is always occluded by the front-facing parts, which are closer to the viewer, and the back-facing faces can simply be skipped. For sharp or polygonal faces, which are supposed to be planar, this amounts to testing the angle between view vector and face normal: Let n_f be the face normal, p_f a point on the face plane, for instance the mid-point of the bounding sphere, and let $v_f = p_f - a_{\text{cone}}$ be the *view vector* from the eye to the point, not normalized. A planar face is front-facing only if $\angle(v_f, n_f) > \frac{\pi}{2} \Leftrightarrow \langle v_f, n_f \rangle < 0$.

Surprisingly, it is possible to also detect back-facing *curved* faces with about the same effort, by using the *normal cone technique*. This method is also treated in some detail by Hoppe [HDD*94], and by Luebke and Erikson [LE97] in the context of triangle meshes. The normal cone measures the variation of the normal vectors over a piece of surface. To determine it accurately, an infinite number of normals on the continuous surface must be evaluated: A normal vector can be considered as a point on the unit sphere (taking vectors for points). The set N_f of all normals from a smooth face f spans a *solid angle*, which corresponds to a 2-cell on the unit sphere. The axis of the view cone is then the vector a where the maximum angle to all other vectors from N_f is minimum, which is $\min_{a \in S^2} (\max_{n \in N_f} \angle(n, a))$. The respective maximum angle is the opening angle of the normal cone.

This can be approximated by taking the limit normal at the limit face point, located near the center of the face, as the direction vector. Then the maximum deviation of all vertex limit normals of the face from the main direction gives an estimate of the normal cone angle. This approximation can be justified by the smoothness of the limit surface and its normal field: Extreme deviations are expected at the face boundary.

The principle of back-patch culling is shown in Fig. 4.42 in a two-dimensional setting. The main face normal n , the limit normal of the face point, is taken as cone axis, and the normal cone angle α_f is the maximum of the angles to the vertex normals. Note that the opening angle of the cone is $2\alpha_f$. This works quite well if the face normal is close to the ‘median’ of the vertex normals (4.42, a). In case the face normal is similar to one of the vertex normals, the opening angle may quickly come close to 180 degrees (4.42, b). The larger the normal cone is, the larger is also the angular region from where the face is visible (4.42, c,d). The view vector $v_f = p_f - a_{\text{cone}}$ is just the reversed direction to the eye. Analogously to the planar case, a curved face is visible if the angle $\angle(v_f, n_f) + \alpha_f > \frac{\pi}{2}$. Since the cosine is decreasing in $[0, \pi]$, and symmetric to the y-axis, this yields a visibility test for curved faces that is just as simple as the test for planar faces:

$$\begin{aligned} \cos(\angle(v_f, n_f)) &< \cos\left(\frac{\pi}{2} - \alpha_f\right) = \cos\left(\alpha_f - \frac{\pi}{2}\right) \\ \Leftrightarrow \left\langle \frac{v_f}{|v_f|}, n_f \right\rangle &< \sin \alpha_f \end{aligned}$$

So the view vector v_f needs to be normalized. The sine of the opening angle can be simply computed as $\sin \alpha_f = \sqrt{1 - \cos^2 \alpha_f} = \sqrt{1 - \langle n_f, n_{\text{max}}^v \rangle^2}$, where n_{max}^v is the vertex normal with the largest angular deviation. Also note that the angle never has to be computed explicitly, since $\langle n_f, n_{\text{max}}^v \rangle = \langle n_f, n^v \rangle_{\text{min}}$. But α_f must be limited to $\frac{\pi}{2}$, i.e., the opening angle may not be more than 180 degrees, otherwise the sine is ambiguous, since $\sin(\frac{\pi}{2} - \alpha) = \sin(\frac{\pi}{2} + \alpha)$.

Depth assignment: The distribution of resolutions. What remains to prepare for tessellation is to assign a subdivision depth to all smooth faces that have passed the culling tests. This reflects a trade-off, since a low resolution gives a bad quality, and a high resolution impairs the rendering performance, as measured in ‘frames per second’ (fps). It depends on both the power of the machine and of the scene complexity. When a minimum frame rate is required, e.g., 20 fps, this determines a *triangle budget*, or, for combined B-reps, an overall *resolution budget* as a sum over all visible faces.

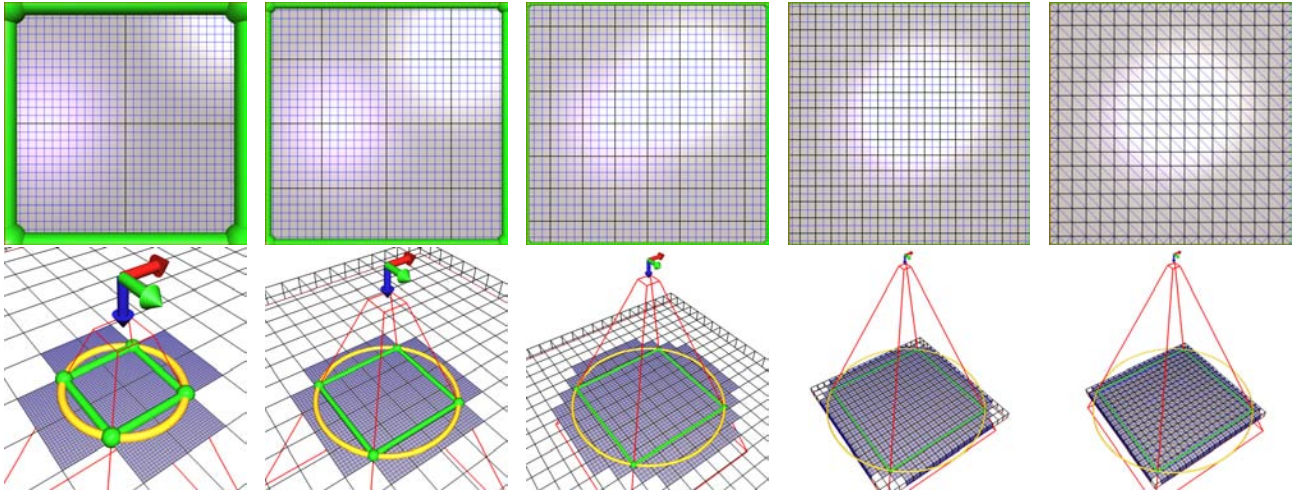


Figure 4.43: The projected size heuristic. With $q = 2.0$ faces with half the diameter of the view cone are shown in full resolution (1a). Whenever the distance is doubled, the resolution is reduced (1b-1d). The screen is covered by the same number of quads, distributed among more faces. With more than 16×16 faces, the depth becomes 0 (1e). Second row: The intersection of the model plane with the view cone (circle) compared to the view frustum.

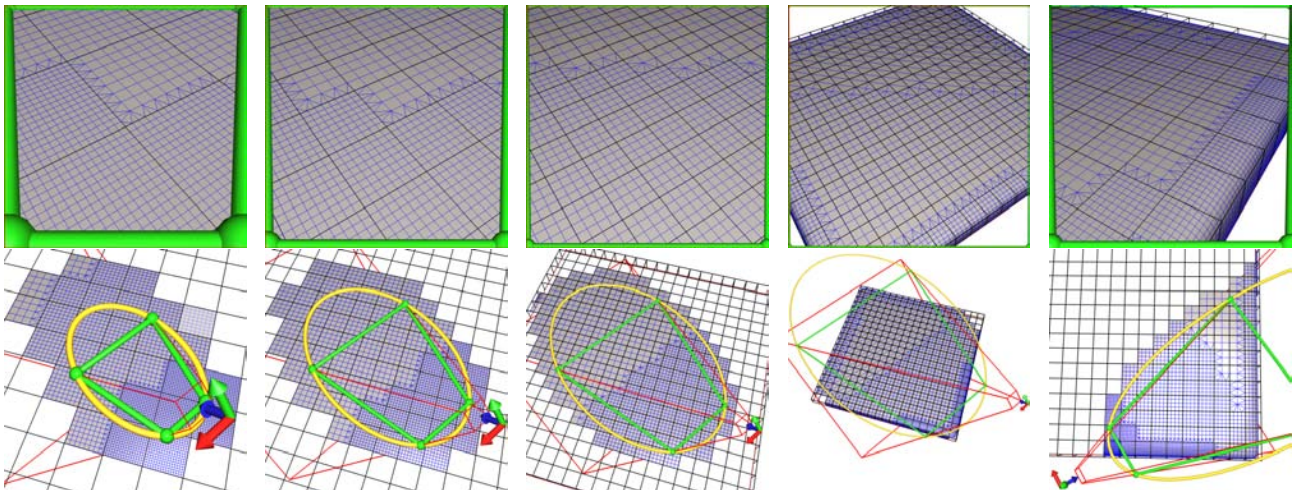


Figure 4.44: Excess of the view cone at grazing angles. Nearby quads have about the same projected size as quads in the back. Second row: At grazing angles, the view cone incorrectly marks many invisible faces as visible (2a-c). The rounded faces have a view cone angle of nearly 45 degrees, so they are subdivided 'earlier' (2d, 2e).

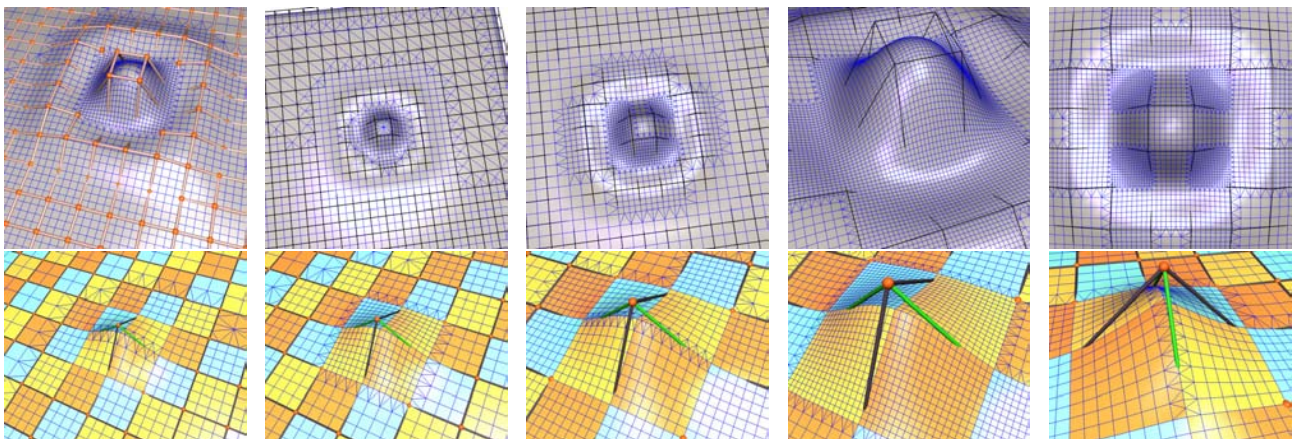


Figure 4.45: The curvature and crease heuristics. For curved faces, the resolution is increased 'earlier' (1a-d). The refinement pattern gives interesting visual insights, showing which faces are more curved than others (1e). Second row: The crease heuristic is activated halfway between one resolution and the next higher resolution.

The question is how to distribute the quality budget among the visible smooth faces. One possibility is to minimize an error measure, for instance the (average) pixel error between two images: The scene rendered at highest resolution is the reference image, and it is compared to an image of a lower resolution scene, obeying the resolution budget. Then the resolution distribution is best which minimizes the image difference. – But unfortunately, the pixel error is not very well adapted to the human perception: for instance the eye is quite sensitive for detecting high color gradients, such as at the object silhouette, or at creases ([Fel92, BFH*98]). Due to these complications, the *visual importance* of a smooth face is ranked according to a set of plausible heuristics, instead of using a strict error metric.

- *Projected size:* All quadrangles of the tessellation should have roughly the same size when projected on the screen. This means that a patch going away twice as far should have its depth decremented by one. A patch with a bounding box diameter twice as large has its depth incremented by one.
- *Curvature:* The angle between normals of neighbouring quadrangles should not exceed a threshold. This means that a face whose normal cone angle is twice as wide should have its depth incremented by one.
- *Silhouette:* Faces which are neither completely front-facing nor completely back-facing are silhouette faces. As a rule of thumb, their depth can be incremented by one.
- *Crease:* Accuracy is incremented by one along a crease: Let d be the maximum of the depth values from the faces on both sides of the crease. For both patches, the *neighbour* depth is then set to $d + 1$, while their actual depth values are not changed.

The projected size of a face can be roughly approximated by the size of the bounding sphere, relative to the view cone. This does not, however, take the orientation of the face into account, but only the size of the bounding sphere r_f , relative to the size of the cone. The latter varies with increasing distance to the viewer: At distance x_f (cf. Fig. 4.41 a), the diameter of the cone is $x_f s_{\text{cone}}$. Consequently, the projected size of the bounding sphere, relative to the view cone, is $p_f = r_f / x_f s_{\text{cone}}$.

The projected size heuristic states the subdivision depth `Face::depth` depends logarithmically on the projected size. So let q be an overall quality parameter that works a threshold: A face is shown at full resolution only when its relative projected size is greater than $1/q$. With $q = 10$ for instance, all faces covering more than 10% of the viewport (in diameter!) receive depth 4. Consequently, faces with projected bounding sphere diameter of more than $1/2q$, $1/4q$, $1/8q$, and $1/16q$ receive depths 3, 2, 1, and 0. The effect of this rule is demonstrated in Fig. 4.43. The depth can be computed with the expression $\lfloor \log_2(q p_f) \rfloor + 4$, since for example $\log_2(q \frac{1}{16q}) = -4$.

The curvature heuristic is realized by using the normal cone. Every smooth face contains $\sin \alpha_f$, which is roughly proportional to the face curvature: It is 0 if the face is flat, $1/\sqrt{2} \approx 0.71$ if the opening angle is 90 degrees (so $\alpha_f = \frac{\pi}{4}$), and 1 if the normal cone degenerates to a half-space when the opening angle is 180 degrees. To integrate it with the projected size heuristic, a higher curvature can be used to virtually increase the projected size, using $q p_f (1 + \sin \alpha_f)$. When feeding this into the logarithm, this increases the resolution of a curved face by one much earlier than for a flat face.

The reason why the projected size is multiplied, instead of adding $\sin \alpha_f$ to the result of the logarithm, is efficiency: One challenging aspect of depth assignment is that it must be fast – because it is performed for each face in each frame. Evaluating \log_2 for each smooth face is inefficient, but it can be avoided using a simple trick: The IEEE floating point format stores numbers as exponent plus mantissa. For 32 bit floats, the exponent is a signed 8-bit integer number in bits 23-30 of the 32 bits, bit 31 is the sign bit of the mantissa. Surprisingly, the projected size and curvature heuristics can both be evaluated with only *two lines* of code, and the crease heuristic with a third line:

```
float q_f = quality * (r_f/(x_f*s_cone)) * (1.0 + face->normalCone);
depth    = (((int&)q_f)>>23)&255)-123;
depthSharp = (((int&)q_f)>>22)&1);
```

For numbers in $[1.0, 2.0)$, the exponent is 127, this is why $-127 + 4 = -123$ is added to the isolated exponent. The depth value must of course then be clamped to the integer range $0 \dots 4$. – Bit 22 is the highest bit of the mantissa, and it is 0, e.g., for numbers in $[1.0, 1.5)$, and 1 for numbers in $[1.5, 2.0)$. So bit 22 can be used to increase the resolution along sharp creases: The crease resolution is increased just before the face resolution is increased. Practical experience shows that this feature should be optional, since it can greatly affect the rendering load from sharp faces. – The silhouette heuristic can be added in a similar fashion as $\sin \alpha_f$, e.g., by using $1 - \langle v_f, v_{\text{cone}} \rangle$ as measure for the silhouette-ness of a face.

Tessellation. All three procedures, view-frustum culling, back-patch culling, and depth assignment, are performed together in the `BRepCombined::determineDepth` function. It is by no means mandatory, however, to use it: The `Face::depth` and `Face::depthSharp` values can be freely set, according to any application-specific metrics or requirements. One possibility, for instance, is to execute `determineDepth` with a low basic quality q , and then to ‘highlight’ more important parts in the surface by assigning a higher resolution. This is especially important when surface properties must be taken into

```

void BRepCombined::tessellate (Face* face)
{
    ... (processing non-smooth faces)
    short depth = face->depth;
    Edge* edge = face->oneEdge;
    short depthLeft;
    short depthTop = edge->mate()->face->depth;
    edge = edge->faceCCW();
    Edge* edgeEnd = edge;
    do {
        depthLeft = edge->mate()->face->depth;
        tessalator.tessellate (edge->patch, depth-1,
                               depthLeft-1, depthTop-1);
        depthTop = depthLeft;
        edge=edge->faceCCW();
    } while (edge!=edgeEnd);
}

```

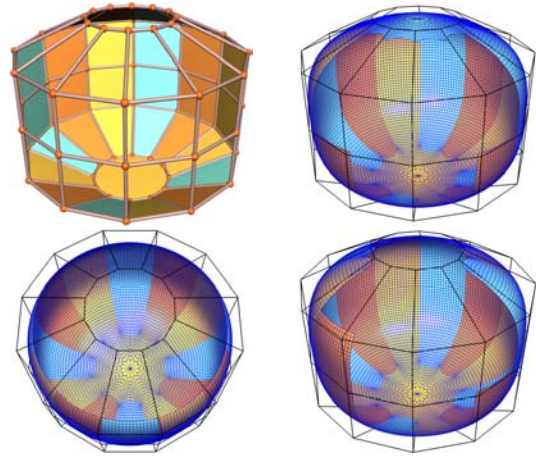


Figure 4.46: Tesselation of smooth faces. Each patch also needs the resolution of two neighbour faces (`depthTop`, `depthLeft`), since each of them may potentially require the patch to refine adaptively. The tessalator does the actual work (see section 3.4.2). The depth values are stored in the patch, so that it can be rendered without further access to the neighbours. Note: To avoid confusion, this example shows only the part concerning smooth faces, and it does *not* show how `depthSharp` is taken into account.

Right: Illustration of back-patch culling using an inverted object with CW front-faces (a). Slightly slanting the object forward (from (b) to (d)) makes the top face vanish. With the normal cone, exclusively front-facing faces can be identified quite accurately, as in the upper part of (c).

account that `determineDepth` can not know about, for instance specular materials that require a high resolution to exhibit faithful highlights, or when transparent materials are used.

Once a resolution is assigned to all smooth faces, `maxDepthSharpFaces` performs one additional, but unavoidable, pass over the sharp faces. It sets the resolution, i.e., `Face::depth`, of visible sharp faces (and of all their rings) to the maximum resolution of its smooth neighbour faces. Sharp faces always have `Face::depthSharp=0`. The `BRepCombined::tessellate` routine iterates over all visible faces to assure the required tessellation exists in the respective caches. The most time consuming processing require the smooth faces.

- For visible **polygonal faces** with missing triangulation, `face->triChunk = -1`, the triangulator computes it, with respect to the mesh vertices, and allocates a chunk of triangles in `BRepCombined::polygonalTriangles`.
- For visible **sharp faces** with missing triangulation, `face->sharpTriChunk[face->depth] = -1`, the triangulator computes it in the required resolution, with respect to the boundary sampling in `face->sharpPtChunk`.
- The boundary of visible **smooth faces** is cyclically traversed and for each patch `TessellateCatmullClark::tessellate` is called with the patch ID, the face resolution, and the resolution of the two neighbour faces adjacent to the patch.

The first two case distinctions are quite lean: Once a polygonal face was visible, and once a specific sharp face resolution was requested, the respective triangulations are computed and stored in the chunks. This is both time and space efficient, since the triangulations are only computed when they are really needed; it may very well be that a resolution 4 triangulation of a sharp face is never requested, in which case it is also never computed.

The treatment of smooth faces is more expensive because it involves halfedge navigation, and one access to the depth of each neighbour face. The reason is that a patch can refine towards a more subdivided neighbour, but in order to do so, the neighbour resolution must at least be known to the patch. The code in Fig. 4.46 shows the tessellation routine of a smooth face schematically. The expensive part is really just the traversal itself, especially when several thousand smooth faces are visible in low resolution. Note that caching is of course also done for smooth faces: As explained in detail in section 3.4, the tessellation grid in each patch (`Patch::vertex`, `normal`) is adaptively refined on demand. Once a particular patch is computed to the required depth, all that the `tessalator.tessellate` call does is to store the current depth, `depthTop`, and `depthLeft` settings in the patch; otherwise, i.e., when `Patch::depthMax` is still smaller than `max(depth, depthTop, depthLeft)`, it computes the subdivision. Note that the depth values are decremented by one: The job of the `TessellateCatmullClark` class starts *after* the first subdivision has partitioned the smooth faces into quadrangular patches.

Rendering. The `BRepCombined::render(Face*)` routine finally renders sharp and polygonal faces just like in the code in Fig. 4.25 each with a single `glDrawElements` call. For smooth faces, the boundary is traversed, and for each halfedge, `TessellateCatmullClark::render(edge->patchID)` is called. Only in case `depth=0` a triangle fan is rendered instead.

4.4 Progressive Combined B-Rep Meshes

Combined B-reps are meshes composed of both free-form and polygonal parts, and their special feature is that they allow on-line manipulation of the control mesh. This is a vital property especially for two purposes:

- **Interactive Mesh Modeling.**

A shape is typically composed and refined by applying shape modeling tools. The parameters of each tool are interactively tweaked and tuned, until, by visual inspection or measuring, the shape matches the specifications and the ideas of the artist. Technically, this implies that the mesh can be restored to the state before the tool was applied, to re-apply the tool with slightly different parameters. This feature is referred to as the *undo capability* of a modeler.

- **Semantic Level-of-Detail.**

Smooth faces are displayed at an adaptive LOD that can be adjusted on a per-face, per-frame basis. But with very large control meshes, this mechanism ceases to guarantee the interactivity, for two reasons: It can not reduce the LOD of polygonal faces, and every face of the control mesh must be processed in every frame. In such cases the control mesh itself must be coarsened. So the solution is to remove detail from the control mesh, and to add it again when it is needed. The removal should happen in a controlled fashion, according to the *model semantics*.

Automatic simplification, as presented in section 4.1.5, turns a triangle mesh into a progressive mesh (section 4.1.6). The level of detail of a progressive mesh can be freely adjusted by traversing the split sequence in either direction. This means that a static shape is turned into a *multiresolution mesh*. One drawback of this approach is that simplification breaks symmetry (cf. Fig. 4.16): the simplification routine has no information about the intended structure of the 3D model, and it has, in case of synthetic shapes, no connection to the modeling history. Consequently, symmetries and regularities in a simplified model are broken, and even a quite regular mesh is turned into a “triangle soup”.

This can be avoided if more control over the split sequence is granted to the user: The application generating a mesh, i. e. the modeler, has got all the information it needs to generate the refinement operations directly from the modeling history. The generated multiple resolutions can also be supervised by the user during the modeling process, to ensure that also a very coarse LOD still exhibits some regularity, and respects the basic structure of the model.

Progressive meshes realize a paradigm change, because they transfer a static model into a sequence of operations. The same is possible for other multi-resolution shape representation methods with a (closed and complete) set of construction operations. The great benefit is that it gives to the creative mind a method for *authoring multiresolution models*.

4.4.1 Euler Operators for Mesh Manipulation

Combined B-reps can of course be manipulated by directly accessing the mesh entities, in conjunction with touching and update, as it was outlined in section 4.3.2. While this may be indispensable for some applications, the drawback is that it can be tedious to assert the mesh consistency in all cases, and to realize the undo capability, if it is needed.

The alternative is to use an operator set with ‘built-in’ consistency, generality, and undo capability, such as the Euler operators from section 2.2, where they have been introduced only in an abstract way (shown in Fig. 2.13). To use them with meshes, they need to be operationalized, i.e., provided with a concrete interface. One way to do this is by a set of functions that exclusively operate on halfedges and entity attributes. As discussed in section 4.1.3, a halfedge represents a unique position in a mesh, i.e., one (vertex,edge,face) combination. Euler operators can be adapted to combined B-reps in a straightforward way: For every mesh entity, there is exactly one Euler operation that creates it. So it is sufficient to augment the Euler operators with additional parameters that specify the entity attributes.

- operations creating an edge are provided with a boolean sharpness flag
- operations creating a vertex are provided with a vertex position of type `Vec3f`
- operations creating a face are provided with a material ID, which is just an integer

This results in a very concise set of 13 *extended Euler operators* for mesh manipulation: Five Euler operators, five inverse operators, and three operators to change the sharpness, position, and material attributes of existing entities (they are self-inverse). The mesh operations are illustrated in Fig. 4.47, and the API, the `BRepProgressive` class, is listed in Fig. 4.48. Note that the parameter order is always crucial when using Euler operators.

The `makeVEFS` operator. A notable difference between the abstract and concrete operator sets is that the latter has no `makeVFS`, but only a `makeVEFS` operator. This is consistent with the B-rep design decision that there is no direct link between vertices and faces, but that they are connected only via halfedges (see 9 and 10 in Fig. 4.21). Both operator sets are compatible, though, since `makeVEFS` is nothing but `makeVFS` followed by `makeEV`. This means that the vertex of the newly created edge lies at position p_1 (Fig. 4.47 1a).

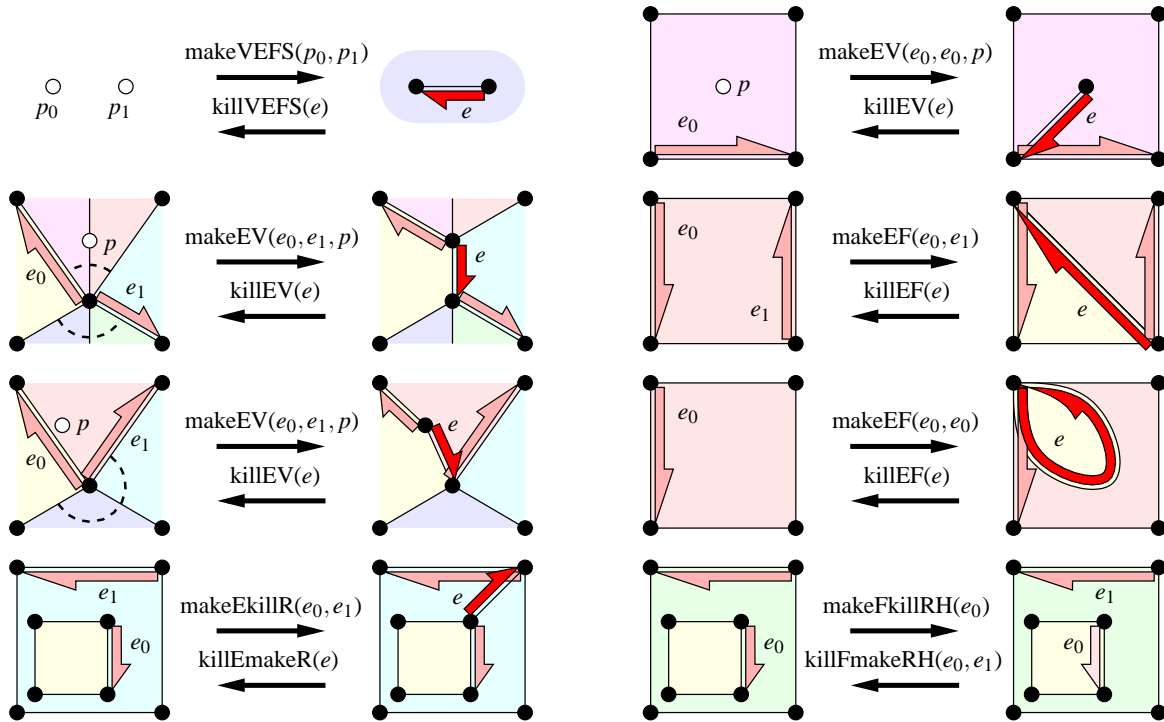


Figure 4.47: Mesh manipulation through Euler operators. The halfedges show how the abstract operators from Fig. 2.13 are operationalized. These functions are part of *progressive combined B-rep meshes* (Fig. 4.48).

The makeEV operator. Its parameters are two halfedges e_0 and e_1 , the position p of the new vertex, and the edge sharpness. Both e_0 and e_1 must be incident to the same vertex v . When both halfedges are identical (Fig. 4.47, 1b), a *dangling vertex* is created. This raises the face degree by 2, since $v = e_0 \rightarrow \text{vertex}$ lies now twice in the face boundary.

When the halfedges are different, the vertex is split. The two halfedges partition v 's edges into two disjoint sets, as indicated by the dotted lines in 4.47 (2a): The sets are $\{e_0, e_0 \rightarrow \text{vertexCW}, e_0 \rightarrow \text{vertexCW} \rightarrow \text{vertexCW}\}$, and analogously for e_1 . The vertex is split so that the edge set of \mathbf{e}_0 is incident to the newly introduced $\mathbf{e} \rightarrow \text{vertex}$. This can also be used for inserting a vertex into an existing edge (4.47, 3a): When $e_1 = e_0 \rightarrow \text{vertexCW}$, the set of e_0 consists of only a single edge, and the newly introduced edge is $e = e_0 \rightarrow \text{vertexCW}$.

Symmetrically, the inverse operator killEV is an edge collapse. It unites the edge rings of two neighbouring vertices.

The makeEF operator. Its parameters are two halfedges e_0 and e_1 , and the material of the face to be created. Both halfedges e_0 and e_1 must be part of the boundary of the same face f . When the halfedges are different, it splits a face in two by inserting a diagonal (4.47, 2b): The two edges partition the face boundary in two disjoint sets $e_0, e_0 \rightarrow \text{faceCCW}$, etc. Note that after the split, e_0 is incident to the **new** face, and e_1 to the **old** face. Also note that e_1 and e share the same vertex, $\mathbf{e}_1 \rightarrow \text{vertex} = \mathbf{e} \rightarrow \text{vertex}$. – When the halfedges are identical, a loop is created in the interior of f .

The inverse operator killEF merges two different faces into one.

The makeEkillR operator. It receives two halfedges e_0 of a ring and e_1 of the ring's baseface, and joins the two boundaries into one (4.47, 3a). The resulting edge e is incident to the ring vertex. So geometrical care must be taken with the inverse operation $\text{killEmakeR}(e)$: e 's vertex becomes a ring vertex. The only check performed is that $\text{killEmakeR}(e)$ can only be applied to an edge with the same face on both sides, i.e., it assures that $e \rightarrow \text{face} == e \rightarrow \text{mate} \rightarrow \text{face}$.

The makeFkillRH operator. It just expects a halfedge of a ring. As usual, it is the simplest, but also the most abstract operation. It converts a ring into a face of its own, which does not change the mesh geometry, but affects only its topology, i.e., the number of holes and/or connected components. In the Fig. 4.47 (3b), to the right, this is expressed by the halfedge in false direction (bright red). This halfedge is part of the *backside* of the quad, which has just become a face of its own. The inverse $\text{killFmakeRH}(e_0, e_1)$ turns e_0 's face (which must not be a ring) into a ring of e_1 's face. So the order here is 'ring first'. If e_0 and e_1 are confused, the result can most likely not be triangulated (ring outside of face boundary).


```

class BRepProgressive
{
public:
    typedef BRepCombined::Mesh Mesh;
    ... etc.

    struct Record { ... see code to the right };
    struct Macro { ... see code to the right };

    BRepProgressive(BRepCombined& cbrep);
    ~BRepProgressive();

// =====
// 1. macro management

    void clear ();
    Macro* newMacro ();
    Macro* clearMacro (Macro* macro);
    void currentMacro (Macro* macro);
    Macro* currentMacro ();
    void groundMacro (Macro* macro);
    void deleteMacro (Macro* macro);

// =====
// 2. mesh manipulation

    Edge* makeVEFS (const Vec3f& p0, const Vec3f& p1,
                  bool s, int mat);
    Edge* makeEV (Edge* e0, Edge* e1, bool s, const Vec3f& p);
    Edge* makeEF (Edge* e0, Edge* e1, bool s, int mat);
    Edge* makeEkillR (Edge* eR, Edge* eF, bool s);
    bool makeFkillRH (Edge* eR, int mat);
    bool killVEFS (Edge* e);
    bool killEV (Edge* e);
    bool killEF (Edge* e);
    Edge* killEmakeR (Edge* e);
    bool killFmakeRH (Edge* eR, Edge* eB);
    bool moveV (Edge* e, const Vec3f& p);
    bool sharpE (Edge* e, bool s);
    bool materialF (Edge* e, int matid);

// =====
// 3. saving edges

    bool edgeToIndex (int& rec, int& macro, int& stamp, Edge* e);
    Edge* indexToEdge (int rec, int macro, int stamp);

// =====
// 4. update, undo and redo.

    bool assertCommitUpdate();
    void undo (Macro* macro);
    void redo (Macro* macro, int depth);

private:
    void undo (Record* rec);
    void redo (Record* rec);

// internal Euler operators do the actual work
    Edge* _makeEV (Edge* e0, Edge* e1, bool s, const Vec3f& p);
    Edge* _makeEV (Edge* e, bool s, const Vec3f& p);
    ...

    int timestamp;
    BRepCombined* cbrep;
    Skipvector<Record> records;
    Skipvector<Macro> macros;
    vector<int> macroDAG;
};

struct Record {
    int& deadID () { ... }
    bool active () { ... }

    short op;
    int sourceId;
    int prev;
    int next;

    int edge0, edge1, edge2;
    int newEdge;
    Vec3f p0, p1;
    bool sharp;
};

struct Macro {
    int& deadID () { ... }
    bool active () { ... }

    int status;
    int firstOp;
    int lastOp;
    int size;
    int timestamp;

    Box3f box;
    Vec3f childSphereMid;
    float childSphereRad;

    set<int> children;
    set<int> parents;
};

BRepProgressive::Edge*
BRepProgressive::_makeEV (
    Edge* eOld, bool sharp, const Vec3f& p)
{
    if (cbrep->mesh->reserve(1,2,0)) {
        cbrep->mesh->relocate(eOld);
    }
    Vertex* vOld = eOld->vertex;
    Face* fOld = eOld->face;
    Vertex* vNew = new_VerTEX(p0);
    Edge* eNew1 = new_Edges(sharp, sharp);
    Edge* eNew0 = eNew1 + 1;

    touch(vOld);
    touch(fOld);

    eNew0->face = fOld;
    eNew1->face = fOld;
    eNew0->vertex = vOld;
    eNew1->vertex = vNew;
    eNew0->next = eNew1;
    fOld->oneEdge = eNew0;
    fOld->oneVertex = vOld;
    vOld->oneEdge = eNew0;
    vNew->oneEdge = eNew1;

    eOld->faceCW()->next = eNew0;
    eNew1->next = eOld;

    return eNew1;
}

```

Figure 4.48: The Progressive B-Rep class.

operation	edges	attributes	#	#ID
$e = \text{makeVEFS}(p_0, p_1, m, s)$	e	p_0, p_1, m, s	5	8 ⁽¹⁾
$\text{killVEFS}(e)$	e	$e \rightarrow \text{vertex} \rightarrow p, e \rightarrow \text{face} \rightarrow \text{material},$ $e \rightarrow \text{mate} \rightarrow \text{vertex} \rightarrow p, e \rightarrow \text{sharp}$	5	8 ⁽¹⁾
$e = \text{makeEV}(e_0, e_1, p, s)$	e, e_0, e_1	p, s	5	6 ⁽¹⁾
$\text{killEV}(e)$	$e, e \rightarrow \text{faceCCW}, (e \rightarrow \text{vertexCW})^*$	$e \rightarrow \text{vertex} \rightarrow p, e \rightarrow \text{sharp}$	5	6 ⁽¹⁾
$e = \text{makeEF}(e_0, e_1, s, m)$	e, e_0, e_1	s, m	5	4 ⁽¹⁾
$\text{killEF}(e)$	$e, (e \rightarrow \text{faceCCW})^*, e \rightarrow \text{vertexCW}$	$e \rightarrow \text{sharp}, e \rightarrow \text{face} \rightarrow \text{material}$	5	4 ⁽¹⁾
$e = \text{makeEkillR}(e_0, e_1, s)$	e, e_0, e_1	s	4	3 ⁽¹⁾
$\text{killEmakeR}(e)$	$e, e \rightarrow \text{faceCCW}, e \rightarrow \text{vertexCW}$	$e \rightarrow \text{sharp}$	4	3 ⁽¹⁾
$\text{makeFkillRH}(e, m)$	$e, e \rightarrow \text{face} \rightarrow \text{baseface} \rightarrow \text{oneEdge}$	m	3	3
$\text{killFmakeRH}(e_0, e_1)$	e_0, e_1	$e_0 \rightarrow \text{face} \rightarrow \text{material}$	3	3
$\text{moveV}(e, p)$	e	$p, e \rightarrow \text{vertex} \rightarrow p$	3	7
$\text{sharpE}(e, s)$	e	$s, e \rightarrow \text{sharp}$	3	1 ⁽²⁾
$\text{materialF}(e, m)$	e	$m, e \rightarrow \text{face} \rightarrow \text{material}$	3	3

Figure 4.49: Data in log records of the extended set of 13 Euler operations, derived from Fig. 4.47.

4.4.2 The Euler Operator Sequence

The execution of the thirteen extended Euler operators can be logged. This creates one log record for each operator, which contains both the *undo*-information, to restore the mesh state before the operation, and the *redo*-information, to re-apply the operation hereafter. The result is a sequence of operations, the *Euler sequence*, very much like the split sequence of progressive triangle meshes.

Euler operators are more general though than the progressive meshes operators: A vertex split creates one vertex, two faces, and three edges, corresponding to the Euler sequence `makeEV`, `makeEF`, `makeEF`. Consequently, a PM split sequence could equivalently be expressed as an Euler sequence, exploiting its invertibility for coarsening and refinement. But an Euler sequence operates at a finer granularity, and it is also more general, in that not only one pair of operations is encoded in the sequence, but all 13 Euler operations can – and sometimes have to – be used for modeling. Examples include the removal of edges between coplanar faces and the deletion of an edge to create a ring using `killEmakeR`.

The log record of an Euler operator. The signatures of the Euler operators and the individual fields of the log records are summarized in Table 4.49, the respective struct `Record` is defined in Fig. 4.48, top right. It is derived from the configurations in Fig. 4.47. The record for an operation op needs to store the construction parameters of both op and its inverse $\text{inv}(op)$: The record for `killEF`(e) must also store the parameters needed by `makeEF` to reconstruct the deleted edge, namely $e \rightarrow \text{faceCCW}$ and $e \rightarrow \text{vertexCW}$. Consequently, every pair of mutually inverse operators has the same number of items in their records, listed in the ‘#’ column of Table 4.49. The $\text{vertex} \rightarrow \text{position}$ is referred to as $\text{vertex} \rightarrow p$ in this table. The last column shows the space needed for the log records: A `Vec3f` with three floats needs three times the space of an ID, which is 4 bytes on 32 bit architectures. The superscript indicates single bits that must be stored for sharpness flags. The records in the actual log are of equal size. They match the union of the signatures of the Euler operators: A log record of type `Record` can hold three edge indices, one material index, two points, and a boolean. Space could be saved by using integers and floats in a union fashion. All log records are stored in the skipvector `BRepProgressive::records`.

There is an issue concerning the deletion of loops and dangling vertices with `killEF` and `killEV`. The entries in 4.49 marked with an * are then set to $e \rightarrow \text{vertexCW}$ for `killEF`, and to $e \rightarrow \text{faceCCW}$ for `killEV`, to duplicate the respective other entry. The inverse `makeEF` and `makeEV` operators then correctly create a loop (4.47, 3b) and a dangling vertex (4.47, 1b). It must also be marked, though, *which* of the two halfedges was deleted. Consider Fig. 4.47 (1b), but with the mate killed instead using `killEV`($e \rightarrow \text{mate}$). The inverse operation must then yield `makeEV`($e \rightarrow \text{mate} \rightarrow \text{mate}$); and analogously for `makeEF` in case of loops. This can be signaled by setting the entries marked with * to `NULL`.

Inverting an Euler sequence, and the sourceID. The inversion of Euler operators is somewhat complicated by the fact that it makes sense to also use operations that actually delete entities. So let $[\dots, (\text{makeEV})_i, \dots, (\text{killEV})_j, \dots]$ be an Euler sequence where operation j kills the edge created by operation i , so $i < j$. To invert the sequence, the operators are inverted and the sequence is reversed, which yields $[\dots, (\text{makeEV})_j = \text{inv}(\text{killEV})_j, \dots, (\text{killEV})_i = \text{inv}(\text{makeEV})_i, \dots]$. Care must be taken that the inverse $(\text{killEV})_i$ kills the right edge, because `makeEV` _{j} probably recreates the edge in a different memory location than before, due to the behavior of the skipvector (see section 4.2.1).

The solution to this problem comes from the observation that every existing edge has a unique original *creator*, i.e., one operation that has created it. The index i of the respective log record is stored in the `BRepCombined::Edge::sourceID` at the time when the edge is created (this eventually demystifies the last combined B-rep data field left to explain). Now for a reference to an edge, not the edge's current array index is stored in the log record, but only the `edge→sourceID`. This applies to all the references in the 'edges' column of table 4.49, only with the exception of edges that are created by a record: When an operator such as $e = \text{makeEV}(\dots)$ creates an edge e , the array index of e is stored in the log record.

So the log record of an edge's creator is the unique place where the edge's current array index is stored.

In the above example, $(\text{makeEV})_i$ creates edge e . Its array index is stored in rec_i , and $e \rightarrow \text{sourceID}$ is set to i . Operation j later in the sequence, $(\text{killEV})_j$, wants to delete e . But before, it stores the edge's `sourceID` i as a reference to e 's creator in its log record rec_j . Now suppose an 'undo' of the sequence is requested, and everything is to be done in reverse. The inverse $(\text{makeEV})_j$ recreates the edge as e' in a different location. But to restore the previous situation, it sets the $e' \rightarrow \text{sourceID}$ to the original creator i , and it writes the new array index of e' back to rec_i as the current location of the edge. Then $(\text{makeEV})_i$ can also be safely undone.

Matters are slightly complicated by the fact that half-edges encode a direction. This issue can be resolved by reserving the least significant bit of the `sourceID` for the distinction between mates. In the example, $e' \rightarrow \text{sourceID}$ is actually not set to i but to $2i$, while the `sourceID` of $e' \rightarrow \text{mate}$, created together with e' , is set to $(2i + 1)$. Accordingly, if operation j is not $\text{killEV}(e')$, but $\text{killEV}(e' \rightarrow \text{mate})$, the array position of $e' \rightarrow \text{mate}$ is written back to rec_i . – Note that when the skipvector `BRepProgressive::records` is purged, the `Edge::sourceID` indices of all edges have to be updated.

Dummy records for externally created edges. A progressive combined B-rep is initialized with a reference to a combined B-rep mesh (see the constructor in Fig. 4.48). This mesh does not have to be empty, and it shall be possible to apply (extended) Euler operators to change it.

The mesh may have been read from a file, or constructed with other modeling methods. In either case, when it is not constructed from scratch with Euler operators, the `sourceIDs` of the edges are invalid, i.e., -1 . There are no source operations to refer to when applying an Euler operator that is to be logged. In this case, a *dummy record* is inserted into the operator sequence for each original edge that is referenced. It serves as a synthetic unique source record for the edge, so that Euler operators can refer to it. Consequently, the index of this record is put into the edge's `sourceID` field. With this extension, all modeling operations can be invertibly applied also to externally created meshes.

4.4.3 Euler Macros and Semantic LOD

The Euler sequence is just one long, unstructured sequence of operations, stored in `BRepProgressive::records`. But one of the goals of progressive combined B-reps is to allow for selective updates: A part of the model should be changeable without having to rebuild the whole sequence. The way this is realized is by introducing a grouping facility for Euler operations, the *Euler macros*.

Every log record belongs to an Euler macro. The Euler macros, of type `Macro`, are stored in the skipvector `BRepProgressive::macros`. Just like halfedges, log records also have a `sourceID` field, which is the index of the macro the log record belongs to. The log records of a macro are contained in a doubly-linked list. It can be iterated through in either direction, in *undo-direction* from `macro.lastOp` following `record.prev`, and in *redo-direction* from `macro.lastOp` following the `record.next` indices. All these log records have the same `sourceID`.

Internal Euler operators and the current macro. Each of the extended Euler operations in the public part of the `BRepProgressive` class (part 2. of the `BRepProgressive` class in Fig. 4.48, left) actually does three things when it is called:

- It executes the respective internal (private) Euler operator `_makeVEFS`, `_makeEV` etc.,
- it appends a properly filled out log record to the record list of the current macro, and
- it checks for dependency between macros (see below).

The *internal Euler operators* do the actual work in the mesh: They allocate entities, set attributes, maintain a valid connectivity, and touch surrounding entities. As one example, the implementation of the internal `_makeEV` operator is given in Fig. 4.48 (lower right). As their main work is to connect pointers, internal Euler operators execute very fast.

The *current macro* is the macro that receives the log record when a public Euler operator is executed. A macro can be made current with the `newMacro`, `currentMacro`, and `clearMacro` calls from `BRepProgressive`. It also is possible to switch between different current macros: The log records have explicit `prev` and `next` references, so a macro's log records do not have to be at successive array positions. With `currentMacro(NULL)`, there is no longer a current macro, and the `BRepProgressive` class switches to *forward modeling* mode (see below).

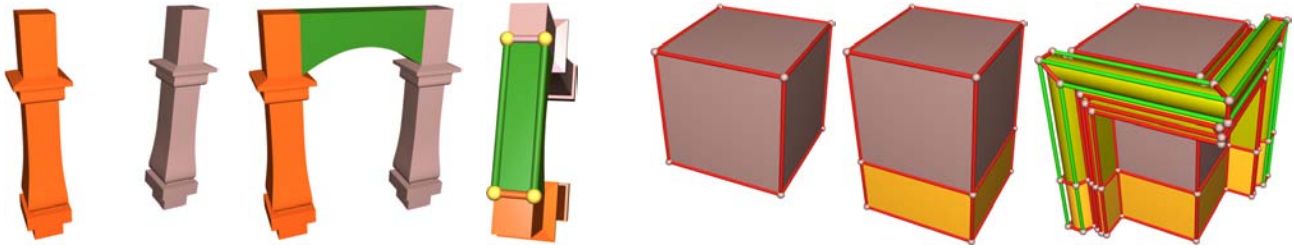


Figure 4.50: The dependency relation of Euler macros. Left: Two columns, each with its own Euler macro, are connected with a third macro to form an arcade. The ‘bridge’ is a child of the column macros, which are its parents. The top face of the bridge has faces that were created by, and thus ‘belong’ to, three different macros. Right: The extrusion is a child of the cube, and the decoration is a child of both the cube and its extrusion. To undo the extrusion, the decoration must be undone first. Cube deletion first deletes the decoration, then the extrusion.

Macro undo and redo. A macro can be in either of two states, active or inactive. A macro is *active* when it was executed, and its edges exist. This is the case right after its creation, or after redo. Undoing a macro puts it in state *inactive*. In order to undo a macro with `undo(Macro*)`, the record sequence is traversed in reverse direction from `lastOp` to `firstOp`, and the private function `undo(Record*)` is called for each record. It executes the respective inverse private Euler operator, with the proper parameters to restore the mesh state before the macro was executed. Symmetrically, for `redo(Macro*)`, the record sequence is traversed from `firstOp` to `lastOp` and the (private) function `redo(Record*)` is called for each record, which in turn calls the internal Euler operators.

The macro dependency graph. There is a canonical dependency relation between macros: Operators that belong to a macro m_A may use halfedges produced by operators from another macro m_B . In this case m_B is called a *child macro* of m_A , and m_A is a *parent macro* of m_B . An undo of m_A will first undo m_B . To redo m_B , first m_A must be redone. So all parents of active macros are also active, and all children of inactive macros are also inactive. The dependency graph induced by the parent-child relation can be regarded, and used, as the *continuation of a scene graph below the object level*.

As the dependency graph is directed and acyclic (a DAG), a partial order exists. It is stored in `BRepProgressive::macroDAG` as a linear array of macro indices. The linear order is such that for any given macro, its parents are located before, and children after its own position in the sequence. The macro DAG is computed from two explicit sets of parents and children contained with each Macro. Macros can be dynamically added (with `newMacro`) and deleted (with `deleteMacro`). In order to completely delete an active macro, first all children are recursively deleted, then the macro itself is undone. Finally, the macro’s records are deactivated, i.e., returned to the skipvector for later reuse.

Note that a single operator can actually change the dependency graph: Each operator using an edge of another macro m' makes the current macro m a child of m' . The examples shown in Fig. 4.50 also demonstrate that in general, neither vertices nor faces belong to only one macro: Both vertices and faces can be incident to edges that were created by different macros. A macro’s Euler operations can in fact be scattered arbitrarily over the whole model.

AssertCommitUpdate and stable edge references. Whenever the mesh is changed, by modeling or by undo/redo of macros, the tessellation of the combined B-rep must be updated to prepare for the rendering (see section 4.3.5). Additionally, after any changes in the dependency graph, the partial order must be recomputed. This is possible in linear time using depth first search [CLR90]. Such update issues are taken care of by the `assertCommitUpdate` function of `BRepProgressive`. It can be used instead of, but also together with `BRepCombined::commitUpdate`.

Macro undo and redo may shuffle the mesh entity arrays. Let m_A and m_B be two independent macros. While a LIFO order such as `undo(m_A); undo(m_B); redo(m_B); redo(m_A)` re-creates entities in exactly the same places, this is not true for the mixed order `undo(m_A); undo(m_B); redo(m_A); redo(m_B)`. This raises the problem of reliably referencing a mesh halfedge.

Recall from the previous section 4.4.2 that an edge’s log record is the place that always contains the current array index of the (full) edge. But to refer to a *halfedge*, the `sourceID` is to be used: It contains the record index, but also encodes a direction in its least significant bit. This alone is still not sufficient, because when a macro is deleted, the record is deactivated for later re-use. Yet the same is also true for the macro itself; and the following sequence may lead to re-using the same record r with a new, different macro that has actually the same index:

```
m = newMacro(); ...(euler operators, 1st version)... deleteMacro(m); m = newMacro(); ...(euler operators, 2nd version)...
```

Let r be the last record of the first version of m . When m is deleted, r is released last, and due to the LIFO behaviour of the record skipvector, it will be the first record allocated by the second version of m . But macros also reside in a skipvector, and for the same reason, m is just at the same location, and has the same index in both versions.

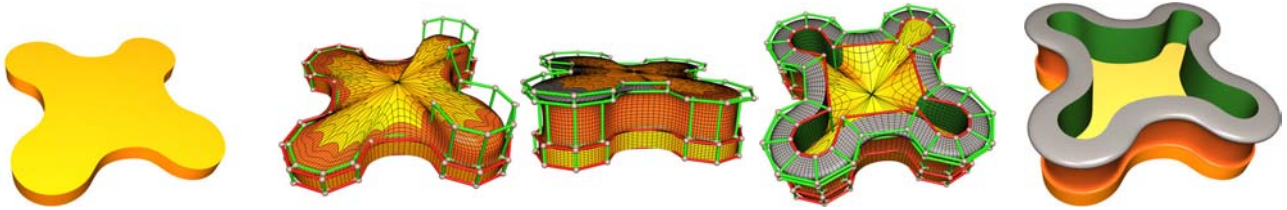


Figure 4.51: Inconsistent intermediate configurations.

The solution is to use a triple index ($sourceID, macroID, timestamp$). Each macro has got a time stamp, which is just an integer in BRepProgressive that is incremented each time `newMacro` and `clearMacro` is called. The timestamp is initialized with a random number at construction time. So a macro can be reliably identified only with a pair of integers ($macroID, timestamp$) to make sure one does not get a re-used version of it. The BRepProgressive provides two functions `edgeToIndex` and `indexToEdge` to convert back and forth between integer triplets and halfedge pointers.

The forward modeling mode, and ‘grounding’ a macro. Sometimes it is favourable *not* to log mesh manipulations: It may be that the user is certain that a manipulation will not have to be un-done. For animations, logging also does not make sense: Animations move mesh vertices incrementally, and in a controlled way. It is not necessary to restore the mesh to its original state each time before applying another animation step.

Logging is switched off by setting the current macro to NULL. This puts the BRepCombined class into forward modeling mode. In this mode, the Euler operators only execute the internal Euler operators, but do not allocate log records; all `sourceIDs` of halfedges created in forward modeling mode are set to -1. Undo is no longer possible, and detail cannot be removed at runtime any more, so the mesh changes are permanent.

It is important to realize that forward modeling can interfere with existing Euler macros: When an edge from an existing macro m is affected by a non-reversible forward modeling action, then m is no longer reversible either. The consequence is that m is *grounded*: Grounding a macro means to activate it, and then to delete it, together with its children. All log records are recursively deleted, and the `sourceID` of all edges created from these records is set to -1. So the effect of a grounded macro is made permanent and can no longer be reversed.

Grounding a macro, with `groundMacro(m)`, may be quite useful, for instance when, after a series of interactive adjustments, a certain type of mesh modification finally is to persist.

Semantic Level-of-Detail. In database terms, an Euler operation is an atomic operation, and an arbitrary sequence of them forms a transaction, which is an Euler macro. Euler macros are the basic unit for undo/redo, unlike PMs, where individual edge-collapse/vertex-splits are the undo/redo unit. The granularity of the macros with respect to the length of the operator sequence is not prescribed. A PM could be emulated by a sequence of Euler macros, each containing just three Euler operations. But Euler macros were developed with a different idea in mind: *Semantic LOD*.

It is based on the observation that experienced modelers often work in a coarse-to-fine fashion: They start with some basic shapes or primitives and successively refine them by adding detail. This modeling style nicely lines up with the macro concept, when a new macro is started every now and then in the modeling process.

The drawback of a low macro granularity is that undo/redo gives popping artifacts. But the advantage on the pro side is that the user – or, synonymously, a higher software layer – can steer the refinement process, and actually author a multi-resolution mesh. It is possible to group arbitrary modeling operations together that belong to the same level of structural refinement. Thus, user-defined macros can be based on the model *semantics* instead of the output of a cost measure for automatic simplification. And in terms of progressive meshes, the edges of a pcB-rep are feature edges – and changing them always produces artifacts, unless the object covers just a few pixels. This is the usual way to hide popping when using LOD. Another reason for a grouping facility is that it helps to avoid geometrically inconsistent intermediate configurations. There is not much use for detail such as a beveled edge or a profile being only constructed halfway. This is illustrated in Fig. 4.51: The user can determine to which construction steps shall be grouped together in a macro. At runtime, the respective detail can be switched on or off.

The window model in Fig. 4.52 is one example of a shape that is modeled using hierarchical refinements. Model hierarchies can be exploited in various ways with macro undo/redo to remove visually unimportant parts:

- Detail size measure, to determine how much (projected) detail a macro adds with respect to its parent
- Macro depth criterion, to unfold the macro DAG just to a fixed depth, such as only to m_1 of all windows
- Occlusion culling, especially important in a city walkthrough scenario, to remove occluded buildings and detail

Such criteria can be used as *undo/redo oracle* for a macro-based LOD, which is also called *macro culling*.

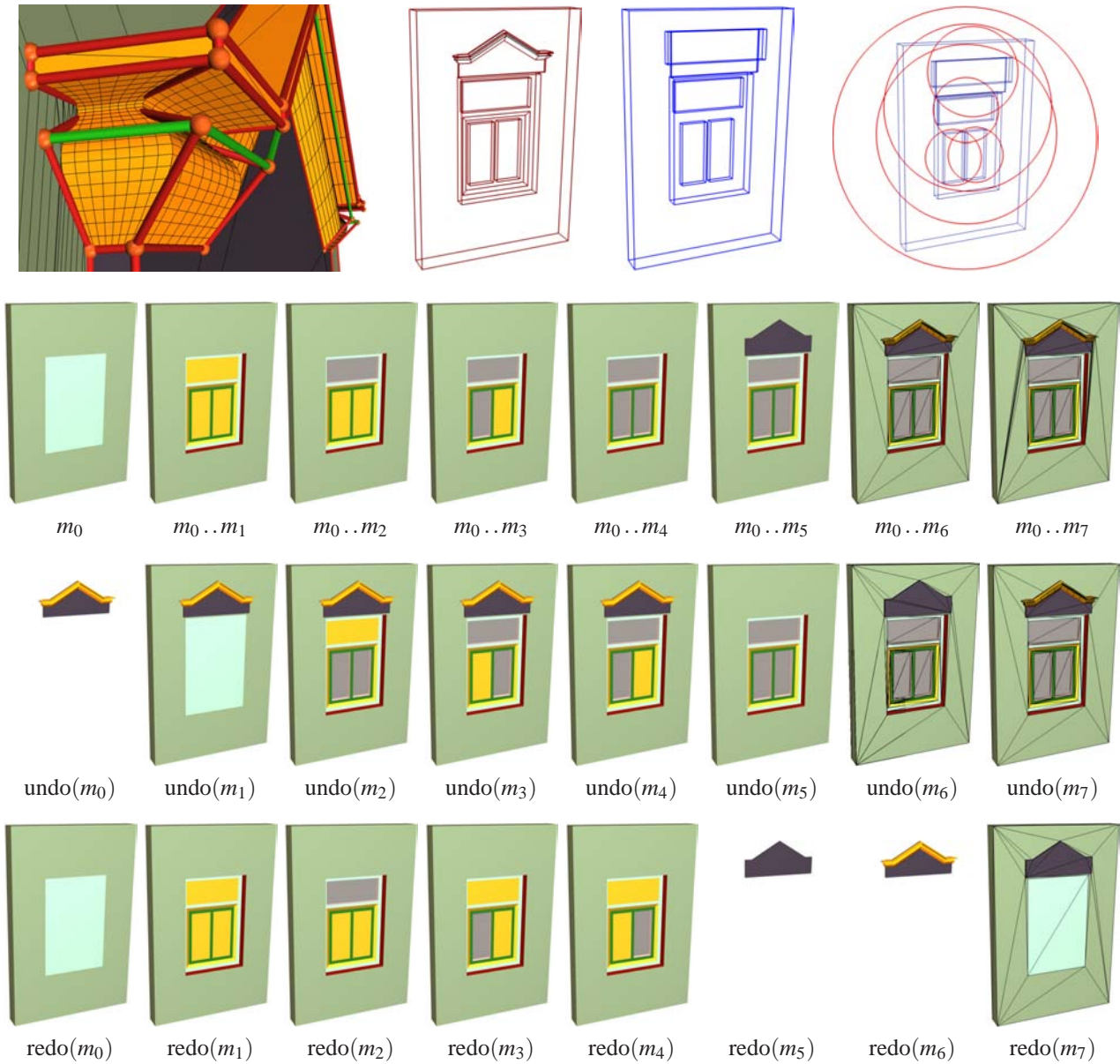


Figure 4.52: Euler macro hierarchy example. Row 1 shows how the window control mesh corresponds to the Euler macro boxes, and to the respective child spheres. The boxes are tight if the model is axis aligned.

Row 2 shows the different steps of the window construction, each step corresponding to one Euler macro: m_0 creates the wall and a quad for the window. This is subdivided by m_1 creating the three yellow fields. These are provided with the actual grey windows by m_2 , m_3 , and m_4 . The decoration in the top is created independently from the wall in two steps by m_5 and m_6 , and finally attached to the wall by m_7 . So m_7 has two parents, m_0 and m_5 , that are independent from each other. Window and decoration have no edges in common.

Row 3 shows what happens if the model is completed (2h), and then one of the macros is switched off. This illustrates the child relationship, because it implies recursive undo of the children. $\text{undo}(m_0)$ turns off all macros, except m_5 and m_6 , because the decoration was created independently. $\text{undo}(m_1)$ does not affect its parent, the wall, but only its children m_2 , m_3 , and m_4 . Undoing them does not affect other macros, because they are leaves in the DAG. m_5 and m_6 can be switched off without affecting the wall. Interestingly, m_7 is not a child of m_6 . $\text{undo}(m_7)$ finally only disconnects the two sub-models, wall/window and decoration.

Row 4 demonstrates the effect of switching off everything, and then only a single macro is switched on again. This illustrates the parent relationship, because it implies a redo of all the macro's direct and indirect parents. m_0 has no parents, but it is parent of m_1 . Yet a redo of m_2 , m_3 , m_4 switches on m_1 and, as a consequence, also m_0 . m_5 and m_6 act independently, and the last image shows that m_7 's parents are indeed only m_0 and m_5 .

4.4.4 Macro Culling

The idea of macro culling is the *semantic magnifying glass*. This is useful in a scenario with a great number of highly detailed models at greater distance, such as a city scene. It does not make sense to render small details such as window frames, ornaments, or door knobs of distant houses. Even if displayed at a low LOD, this slows down the rendering process intolerably; and combined B-rep LOD does not help with polygonal detail at all. So the goal is to switch off Euler macros that do not significantly contribute to the rendered image. This requires again to formalize the concept of visual importance, and to identify a set of plausible heuristics, like in section 4.3.5 for the depth assignment of smooth faces.

The undo/redo oracle. For macro culling, this amounts to finding an oracle that, given the eye position, tells whether a particular Euler macro is to be switched on or off. This decision should be stable, i.e., hold for a number of frames, to prevent the renderer from having to undo and redo Euler macros all the time. Recall that whenever the mesh has changed, the `commitUpdate` routine from section 4.3.5 must restore the tessellation. `CommitUpdate` is fast, but considerably slower than the LOD based on depth assignment. The machine performance determines an *undo/redo budget*, the maximum number of changes that can be made to a given mesh without violating the mandatory frame rate of 20 fps.

One consequence is that the oracle should not take the view direction into account. The view direction usually varies much faster than the position of the viewer, considering for instance the city walkthrough scenario. High detail should be present in the vicinity of the viewer, gradually decreasing with greater distance. Size plays a rôle as well, which again brings in the solid angle as heuristic measure. And finally the oracle should be compatible with the recursive structure of Euler macros: It makes no sense if the oracle tells to redo a macro, but to undo one of its parents. An oracle that never simultaneously declares a child visible, and one of its (direct or indirect) parents invisible, is said to *respect the macro parent-child relation*.

The active front. In articles from Hoppe [Hop97] and Luebke [LE97] the active tree technique has been proposed for view-dependent refinement of progressive triangle meshes. It is based on the observation that the dependency graph of vertex splits is a binary tree: When a vertex is split, two new vertices are created, which can be split as well. The vertices that are present in a given resolution form the *active tree*. The *active front* are the active vertices with inactive children, i.e., vertices that have not been split, but may be split. Since vertex unfolding starts from the coarse base mesh, the active front is the boundary between the active tree, near the root, and the yet unfolded high detail near the leafs of the vertex split tree. In every frame, it is sufficient to test only the vertices from the active front, to decide whether they should be removed (collapsed), or further expanded.

Literally the same is possible with Euler macros. The active front contains all active macros that have inactive children. Macros that neither have children nor parents are always part of the active front. In every frame the oracle is evaluated for each macro in the active front. To resolve the stability issue, the oracle returns a float value instead of a binary decision. This value is tested against two threshold values s_{undo} and s_{redo} .

- If the oracle value of an active macro, i.e., its importance, drops below s_{undo} , it is recursively deactivated (*undo*). In this case, if the macro has parents, they are added to the active front, and the macro itself is removed from it.
- If the oracle value of an inactive macro has become greater than s_{redo} , it is activated (*redo*), and added to the active front. When all children are active, the macro is removed from the active front.

The thresholds are chosen such that $s_{\text{undo}} < s_{\text{redo}}$. For Euler macros whose importance is between s_{undo} and s_{redo} , this means that they keep their status: if they are active, they remain active, and if they are inactive, they remain inactive.

The sphere tree oracle. This is a first attempt to realize a simple oracle. For every Euler macro, a region of influence is maintained, which is an axis-aligned bounding box (or *AABB*) of type `Box3f`, represented by two 3D vectors (of type `Vec3f`) as $(x_{\text{min}}, y_{\text{min}}, z_{\text{min}})$ and $(x_{\text{max}}, y_{\text{max}}, z_{\text{max}})$. It is stored in `Macro::box`. The box includes the vertex positions of every halfedge referenced by an operator from the macro's log record sequence. Additionally, every macro contains a bounding sphere, the *child sphere*. It is recursively defined as containing the macro's bounding box, and the bounding spheres of the macro's children. Due to the complexity of the enclosing sphere problem [dvOS97], it is computed in a simplistic fashion from the macro's bounding box, which is expanded to contain all child spheres.

The sphere tree oracle simply returns the projected size of a macro's child sphere, relative to the viewport. So it uses the same type of calculations as in the projected size heuristic for depth assignment, presented in section 4.3.5. This oracle respects the parent-child relation: If m_B is a direct or indirect child of m_A , then the child sphere of m_B is contained in the child sphere of m_A . So from all directions the projected size of the latter is at least the projected size of the first.

Some results obtained by the sphere tree oracle are shown in Fig. 4.53. For demonstration purposes, they deliberately use undo/redo thresholds that are quite large. To efficiently avoid popping artifacts, the thresholds should be much smaller.

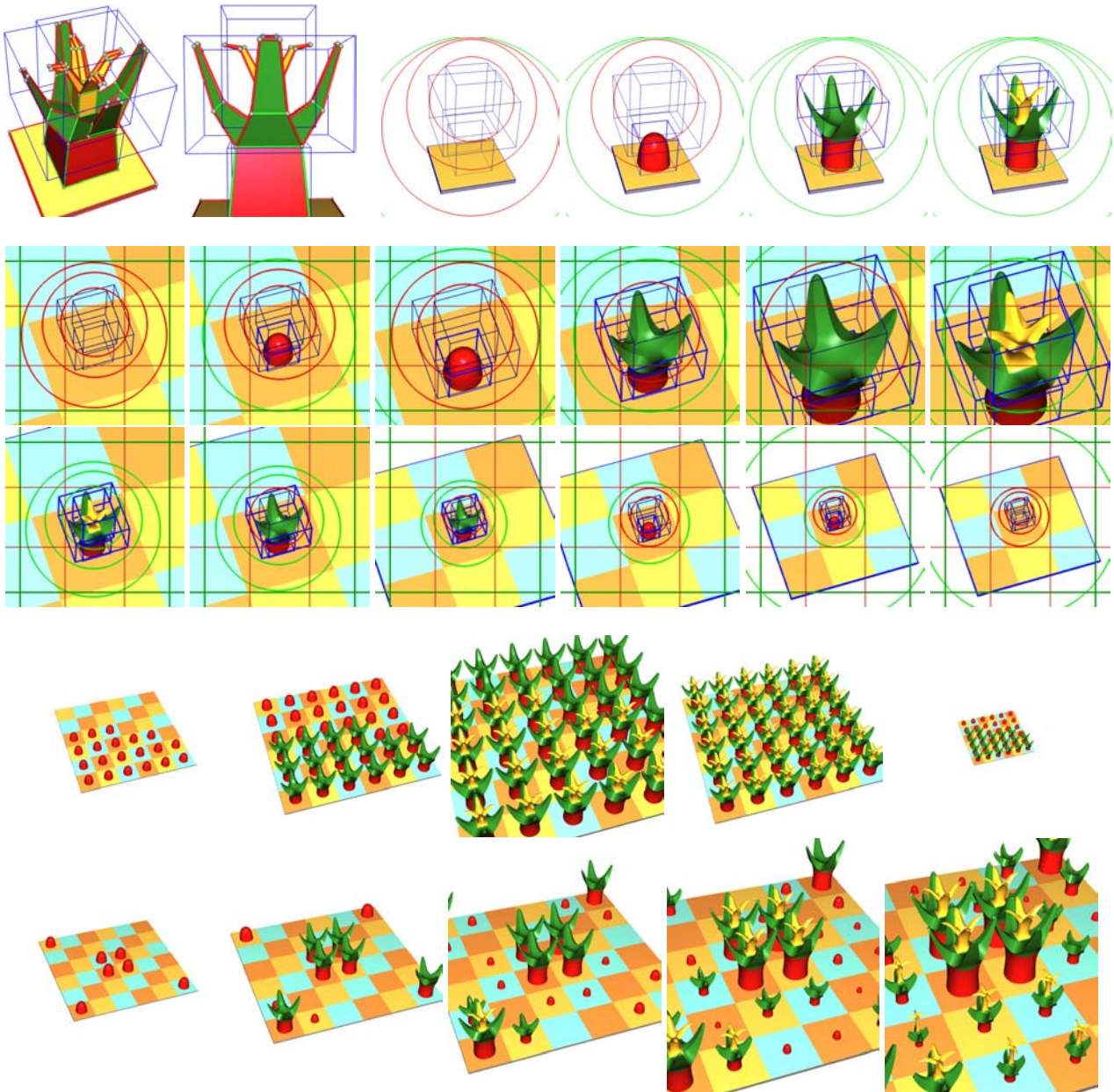


Figure 4.53: Macro culling example with large undo/redo thresholds. Row 1: Each flower-like object is made of three macros m_0 (red), m_1 (green), and m_2 (yellow). The DAG is just a linear sequence. The child sphere of m_2 encloses the macro box tightly, and the child sphere of m_0 is biggest, since it encloses those of m_1 and m_2 . The circle representing the child sphere is green if the macro is active, and red otherwise.

Row 2: The user is approaching an inactive object. The spacing of the green grid represents the activation threshold s_{redo} . Whenever the projected size of a child sphere exceeds the threshold, an inactive macro is activated.

Row 3: The user recedes from an active object. The spacing of the red grid represents the de-activation threshold s_{undo} . Whenever the projected size of a child sphere drops below this threshold, an active macro is deactivated.

Row 4: When approaching a grid of inactive equal-sized flower objects, the projected child sphere heuristic boils down to a distance heuristic. For each macro type m_0 etc., the activation thresholds turns into a distance threshold: When inside a certain distance interval $[dist_{undo}, dist_{redo}]$, an inactive macro remains inactive, and an active macro remains active.

Row 5: When the flowers have different sizes, the undo/redo behaviour is less uniform. Larger flowers are activated to full detail much earlier than small flowers. They are also tessellated at a higher level of detail due to the projected size heuristic of the combined B-rep's depth assignment (section 4.3.5).

4.5 Mesh Modeling Tools

The progressive combined B-rep class `BRepProgressive` provides only the low-level Euler operators as an interface for mesh creation and manipulation. They allow to insert individual edges and vertices, to split faces, etc. They are not suitable as end user interface, e.g., for artists. Instead, end user tools should be implemented on top of the Euler operators. This section presents some examples of more complex modeling tools that hide away the intricacies of Euler operators.

Modeling tools must maintain geometric consistency. Euler operators maintain the mesh connectivity consistent. This means that the abstract 2-complex is consistent in the sense of the mesh definition 2.30: One can only be sure that an embedding *exists* that is also geometrically consistent. Yet Euler operators – deliberately – do not check for geometric consistency. The reason is that it cannot be decided on the level of Euler operators. During modeling, intermediate configurations can occur that are not consistent. One example was given in Fig. 4.51. It produces heavily non-planar faces during the construction process, whereas the end product contains only planar faces. Another example is Fig. 4.46 with an object that is inside out for demonstration purposes. So the notion of consistency in general depends on the intended purpose of a model. When designing mesh modeling tools, two different approaches are possible to ensure geometric consistency. The second approach is more flexible, but the first approach may introduce less overhead:

- to build consistency checks into each tool, so that a tool either refuses to work with infeasible input, or it even tries to ‘repair’ the input, or
- to create inspection tools that check different aspects of an existing mesh, for instance a tool that make a set of faces ‘as planar as possible’.

4.5.1 Converting a polygon into a double-sided face

For practical modeling, the point of departure when creating an object is most often not just a single edge with a degree 2 face (the result of `makeVEFS`), but a higher degree double-sided face. The `poly2doubleface` operator converts a polygon, given simply as a sequence of n 3D points, to a new shell with n vertices, n edges ($2n$ halfedges), and two faces. Formally, the `poly2doubleface` operator is a *complex operator*.

Definition 4.6 (Complex modeling operator, tool)

A *complex operator*, also called a *modeling tool*, is a modeling operation that uses a parameterized sequence of other modeling operations to create a shape. These modeling operations can be either also complex operators, or they are elementary operators.

This notion of a modeling tool is very general, and applicable to all shape representation methods that provide a set of elementary shape construction operators. For the combined B-reps representation, these are the 13 extended Euler operators. The double-sided face is created using a slight generalization of steps 1, 2, 3 from the quad torus example in Fig. 2.15, i.e., by $1 \times$ `makeVEFS`, $n \times$ `makeEV`, and finally $1 \times$ `makeEF`.

The code example for `poly2doublefaceSIMPLE` is shown in Fig. 4.54. The input consistency is only checked in so far as the polygon must contain $n \geq 3$ points p_0, \dots, p_{n-1} (line 7). As soon as the output size is known (or can be estimated conservatively), the mesh entities that will be allocated must be reserved in advance (line 8). This is important to avoid mesh relocations *during* the operator execution (in the loop), which would invalidate, e.g., the e_0 pointer (see sections 4.1.2 and 4.2.1). Line 10 begins the new shell with `makeVEFS`, which is just `makeVFS` followed by `makeEV`. The vertex position of the resulting edge is p_1 , and its mate (at p_0) is saved as e_0 for `makeEF` (line 16). The final `faceCCW` assures that the halfedge returned by `poly2doubleface` is always incident to the vertex at position p_0 from the input polygon. Such a deterministic behaviour is most desirable when the returned edge is to be used by other modeling tools.

Dealing with ill-defined input. Modeling tools should be designed such as to provide maximal flexibility within a well-defined specification, related to all possible configurations of the input variables. In particular, this applies to ill-defined input configurations that can sometimes be made to produce very particular output configurations. The input of `poly2doubleface` should be a simple, i.e., non-intersecting, planar polygon. The planarity, though, is not checked, and consequently, self-intersections can in general not be detected either. This decision is based on the following arguments:

- In case one *knows* the polygon is planar, checking for planarity is an overhead.
- When the polygon is not planar, this may be intentional: Non-planarity is not a problem, e.g., with smooth faces.
- The polygon may have to be projected to a face plane, but there are several ways to determine such a plane.
- If the polygon is planar but intersects itself, it is not clear how to resolve that situation.

So `poly2doubleface` is specified quite modestly as to process the input polygon ‘as it is’, assuming it has been processed so that it conforms to the user’s requirements.

```

1 Edge* poly2doublefaceSIMPLE(BRepProgressive& pcbrep,
2                             const Vec3f* p,
3                             const Vec3f* pEnd,
4                             int sharpmode)
5 {
6     int n = pEnd-p;
7     if(n<3) { return NULL; }
8     pcbrep.mesh().reserve(n, 2*n, 1);
9     bool sharp = (sharpmode==1);
10    Edge* e = pcbrep.makeVEFS(p[0], p[1], sharp);
11    Edge* e0 = e->mate();
12
13    for(p+=2; p!=pEnd; ++p) {
14        e = pcbrep.makeEV(e, e, sharp, *p);
15    }
16    return pcbrep.makeEF(e0, e, sharp)->faceCCW();
17 }

```

Figure 4.54: The simple version of poly2doubleface. It does not check for multiple vertices. The sharpmode parameter determines the sharpness of the edges created.

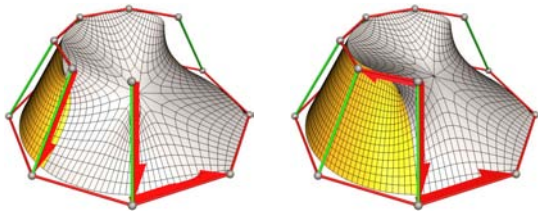


Figure 4.55: Extrusion illustrated. The makeEV and makeEF work such that the newly created face is a side quad. Note that in this example the face is also shrunk.

```

1 Edge* extrudeSIMPLE(BRepProgressive& pcbrep,
2                    Edge* edge, const Vec3f& dir,
3                    int sharpmode)
4 {
5     bool vSharp = ((sharpmode&1)==1);
6     bool hSharp = ((sharpmode&2)==2);
7     int edgeid = pcbrep.index(edge);
8     int n = edge->face->degree();
9
10    pcbrep.mesh().reserve(n, 2*2*n, n);
11    edge = pcbrep.edge(edgeid);
12    Vec3f p = edge->vertex->position + dir;
13    Edge* e0 = pcbrep.makeEV(edge, edge, vSharp, p);
14    Edge* e1;
15    Edge* eEnd = e0->mate();
16    edge = edge->faceCCW();
17
18    do {
19        p = edge->vertex->position + dir;
20        e1 = pcbrep.makeEV(edge, edge, vSharp, p);
21        pcbrep.makeEF(e0, e1, hSharp);
22        e0 = e1;
23        edge = edge->faceCCW();
24    } while (edge!=eEnd);
25
26    eEnd = eEnd->faceCCW();
27    pcbrep.makeEF(e0, eEnd, hSharp);
28    return eEnd;
29 }

```

Figure 4.56: The simple version of an extrusion. The same displacement vector dir is added to all vertex positions of the existing face to create the new vertices.

```

1 Edge* bridgeFacesSIMPLE(BRepProgressive& pcbrep,
2                         Edge* e0,
3                         Edge* e1, int sharpmode)
4 {
5     if ( e0->hasRings() || e1->hasRings() ||
6         !e0->face->isBaseface() || e0==e1 ||
7         !e1->face->isBaseface() || ) { return NULL; }
8
9     int n = 0;
10    Edge* eA = e0;
11    Edge* eB = e1;
12    do {
13        eA = eA->faceCCW();
14        eB = eB->faceCCW();
15        n++;
16    } while (eA!=e0 && eB!=e1);
17    if (eA!=e0 || eB!=e1) { return NULL; }
18
19    bool sharp = (sharpmode==1);
20    int e0id = pcbrep.index(e0);
21    int e1id = pcbrep.index(e1);
22    pcbrep.mesh().reserve(0, 2*n, n);
23    e0 = pcbrep.edge(e0id);
24    e1 = pcbrep.edge(e1id);
25
26    pcbrep.killFmakeRH(e0, e1);
27    eA = pcbrep.makeEkillR(e0, e1, sharp)
28    eA = eA->faceCW();
29    eB = e1->faceCCW();
30    while (eA!=e0) {
31        eA = pcbrep.makeEF(eA, eB, sharp);
32        eA = eA->mate()->faceCW();
33        eB = eB->faceCCW();
34    }
35    return eB;
36 }

```

Figure 4.57: The simple version of bridgeFaces. A pair of different faces with the same degree is connected by a bridge, or a tunnel, made of quads. Input edge pointers must be converted to indices since reserve may issue an array relocation.

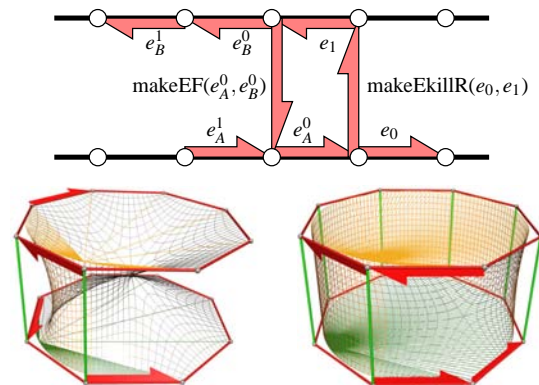


Figure 4.58: Bridging two faces. The first connecting edge is created with makeEkillRH(e_0, e_1), and e_A and e_B traverse the face boundaries of the faces of e_0 and e_1 in CW and CCW directions.

More flexibility with temporary vertex flags. One type of infeasible input configurations can be detected, though: successive polygon vertices that are at the same position, also called *multiple vertices*. They must be removed to avoid zero-length edges. But multiple vertices can be used for a purpose, namely to specify corners in the resulting face: The vertex type of a multiple vertex is set to `CornerVertex`, whereas singular vertices are flagged as `SmoothVertex`. This rule is inspired by B-spline curves, where multiple knot values can also be used to define a smooth curve with corners.

The vertex type assignments are only temporary, though: The actual vertex classification (see Fig. 4.32) is done in `pcbrep.assertCommitUpdate()` (Fig. 4.48), or `cbrep.commitUpdate()` (Fig. 4.33), respectively. Note that the vertices produced by `poly2doubleface` have valence 2, so they can actually never be corners if `sharpmode = 0` in the function from Fig. 4.54. But the update functions are only executed *after* a sequence of modeling operations, just before rendering. So the temporary vertex flags can in fact serve as ‘hints’ for other modeling operations in the same sequence which further modify the double sided face that was just produced.

The extended `poly2doubleface` routine uses the `sharpmode` parameter to specify the temporary vertex type setting:

- `sharpmode = 0` or `1` : The vertex type is set to `SmoothVertex` for all vertices
- `sharpmode = 2` or `3` : The vertex type is set to `CornerVertex` for all vertices
- `sharpmode = 4` or `5` : The vertex type is set according to the multiplicity of the polygon vertex
- `sharpmode = 6` or `7` : The vertex type is not set

Bit 0 of `sharpmode` specifies the sharpness of the edges created, just as in the simple version of `poly2doubleface` from Fig. 4.54. Note that in case the double-sided face is not further modified, the vertex type settings have no effect.

4.5.2 Bridging two Faces

`BridgeFaces` is another complex operator that creates a *bridge*, or a *tunnel*, between two faces f_0 and f_1 that have the same degree. A bridge is created if f_0 and f_1 are facing each other. A bridge can be either a connection between two different shells, merging them into one, or it creates a topological hole when the faces belong to the same shell (as in Fig. 2.29 (d) from chapter 2). When the faces are facing away from each other, `bridgeFaces` also creates a topological hole. But this time the hole goes through the object’s interior, so it is called a tunnel. When both faces are facing away from each other, but belong to different shells, the result is a surface self-intersection. So the `bridgeFaces` operator changes the number, or the genus, of a mesh: It either decrements the number of shells, or it increments the number of topological holes by one. Both effects can be achieved by literally the same sequence of Euler operators. Note that `bridgeFaces` is purely topological in nature, and it does not involve any geometric computations.

The code of `bridgeFacesSIMPLE` is shown in Fig. 4.57. It realizes only a very simple consistency check: f_0 and f_1 must be different baselaces without rings (lines 5-6) and with the same face degree n (lines 8-16). Just like in the previous section, it is wise to reserve $2n$ halfedges and n faces in advance, to prevent mesh relocations during the loop. A relocation, however, may happen as the result of the `reserve` call. Therefore, the two input halfedge pointers e_0 and e_1 must be converted to edge indices before `reserve`, and converted back to pointers after it (lines 19-23). Note that it is not necessary to use stable edge references here (see section 4.4.3) :

- stable edge references are index triplets that survive (i.e., remain valid) arbitrary macro undo/redo actions
- entity indices, in particular edge indices, survive relocation

It is advisable to store all references that are supposed to remain valid for a longer time as stable edge references. References that are to be used during the whole modeling process (between two `commitUpdate` calls) should be stored as indices, and the inner loops should use pointer references for faster access.

The actual bridge is created in lines 25-33. Crucial is the beginning, with the sequence `killFmakeRH`, to make $f_0 = e_0 \rightarrow \text{face}$ a ring of $f_1 = e_1 \rightarrow \text{face}$, and `makeEkillR`, to immediately kill the ring with a first connecting edge (lines 25-26, and Fig. 4.58). The other vertices are connected, a pair at a time, with `makeEF` creating quad faces: e_A traverses the boundary of f_0 in CW direction, and e_B correspondingly f_1 in CW direction. Note that e_A is derived from the return value of `makeEF` by `mate() \rightarrow \text{faceCW}()`. It might as well use `vertexCCW() \rightarrow \text{mate}()`, which is faster if the vertex valence is smaller than the face degree: Neither `vertexCCW` nor `faceCW` are constant time operations (Fig. 4.21, no. 4). – The edge returned by `bridgeFaces` is the mate of the first edge, created by `makeEkillR`. It is incident to the vertex of input edge e_1 .

More flexibility with temporary vertex flags. A more advanced version of `bridgeRings` can take the vertex flags into account that might have been created, e.g., by `poly2doubleface` from the last section.

- `sharpmode = 0` or `1` : The edge is sharp only if `sharpmode = 1`
- `sharpmode = 2` : The edge is sharp only if both $e_A \rightarrow \text{vertex}$ and $e_B \rightarrow \text{vertex}$ are corners
- `sharpmode = 3` : The edge is sharp only if $e_A \rightarrow \text{vertex}$ or $e_B \rightarrow \text{vertex}$ are corners
- `sharpmode = 4` : The edge is sharp only if $e_A \rightarrow \text{vertex}$ (from f_0) is a corner

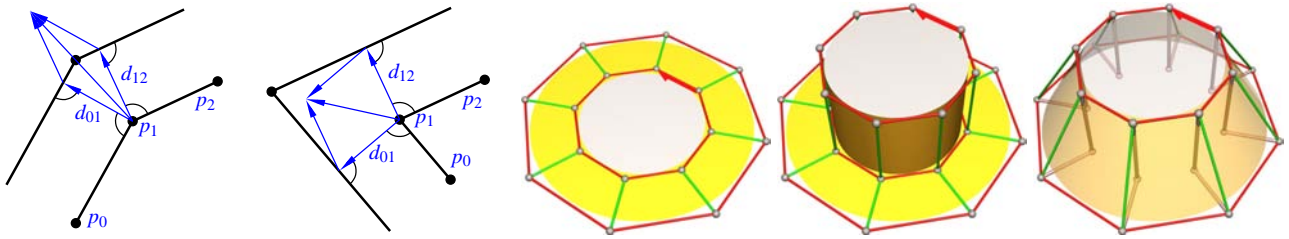


Figure 4.59: Computing the vertices of the offset polygon. The offset vertex position p'_1 may be computed as the intersection of two lines parallel to the segments $\overline{p_0p_1}$ and $\overline{p_1p_2}$ in distance d .

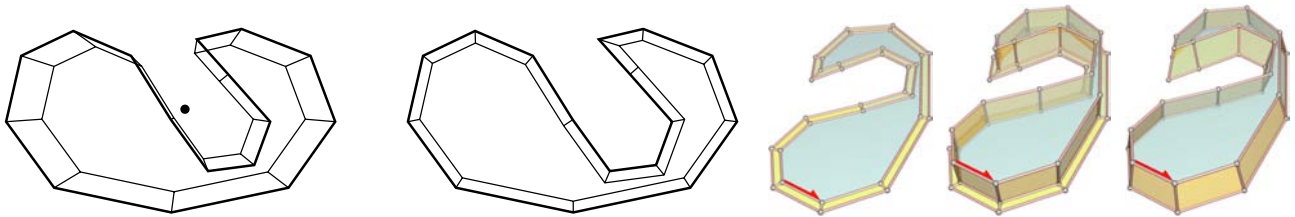


Figure 4.60: Polygon scaling vs. edge offsets. Difference between scaling (a) with respect to a pivot point and horizontal edge offset (b) becomes especially apparent with a non-convex polygon.

4.5.3 A Simple Extrude Tool

Extrusion is a very versatile modeling tool with a variety of different incarnations. In its simplest form, to extrude a horizontal, planar face means to duplicate it, and to gradually move the duplicate in vertical direction. During displacement, each boundary edge ‘sweeps out’ one quadrangular side face. Each of these *side quads* is formed by one boundary edge, its displaced duplicate, and the two vertical edges swept out by the boundary edge’s end vertices.

Another way to look at extrusion is to convert a face to a polygon and to displace the polygon in normal direction. The displaced polygon is converted to a double-sided face, and the `bridgeFaces` operator from the previous section is applied, just as it is shown in Fig. 4.58. The application of `bridgeFaces` also yields a number of quadrangle faces as the side quads of the extrusion. This basic idea can be varied in several ways.

- It can be generalized in a straightforward fashion to an extrusion of faces with holes. Conceptually, this works using `bridgeFaces` to create tunnels for the extrusions of the rings.
- The shape of the extruded face can be varied. As an example, it is possible to move the copied vertices not only in vertical (i.e., face normal) direction, but also in horizontal direction, i.e., within the (displaced) face plane.
- Vertices may in fact be put in arbitrary positions, which may especially be useful with non-planar smooth faces – and this may also be the only option in cases where the face to be extruded is not planar, so that the normal direction is not well defined.

The code for the simplest version of an extrusion is shown in Fig. 4.56. It does not use any consistency checks, in particular it does not check whether the face has any holes. This sounds dangerous, but can in fact even be useful. The reason is that on the B-rep level the extruded face is identical to the original face: The side quads are each created by performing `makeEV`, `makeEF` – and the `makeEF` is oriented such that the newly created face is the quad. This can be nicely seen in Fig. 4.55, which also demonstrates horizontal vertex displacement (shrinking).

So in the end, the outer boundary of the extruded face has the same degree as before, and its rings still belong to it. So when a planar face with rings is extruded, geometric consistency with the rings can be restored by either extruding them as well, or by just moving them geometrically. So also this very simple version of extrude is a versatile tool.

Horizontal edge offsets. There are many ways to shift the vertices horizontally, e.g., within the displaced face plane. One possibility is uniform or non-uniform scaling with respect to some pivot point c , for instance the centroid, of the extruded face. For a displaced vertex $p'_v = p_v + d \cdot n$, with n the face normal, uniform scaling by a factor of s is realized as $p''_v = c + s(p'_v - c)$. It appears though that scaling a face is not very useful when constructing real-world objects. A different rule is used much more frequently, the offset rule: Just as the edges of the scaled polygon, offset edges are also

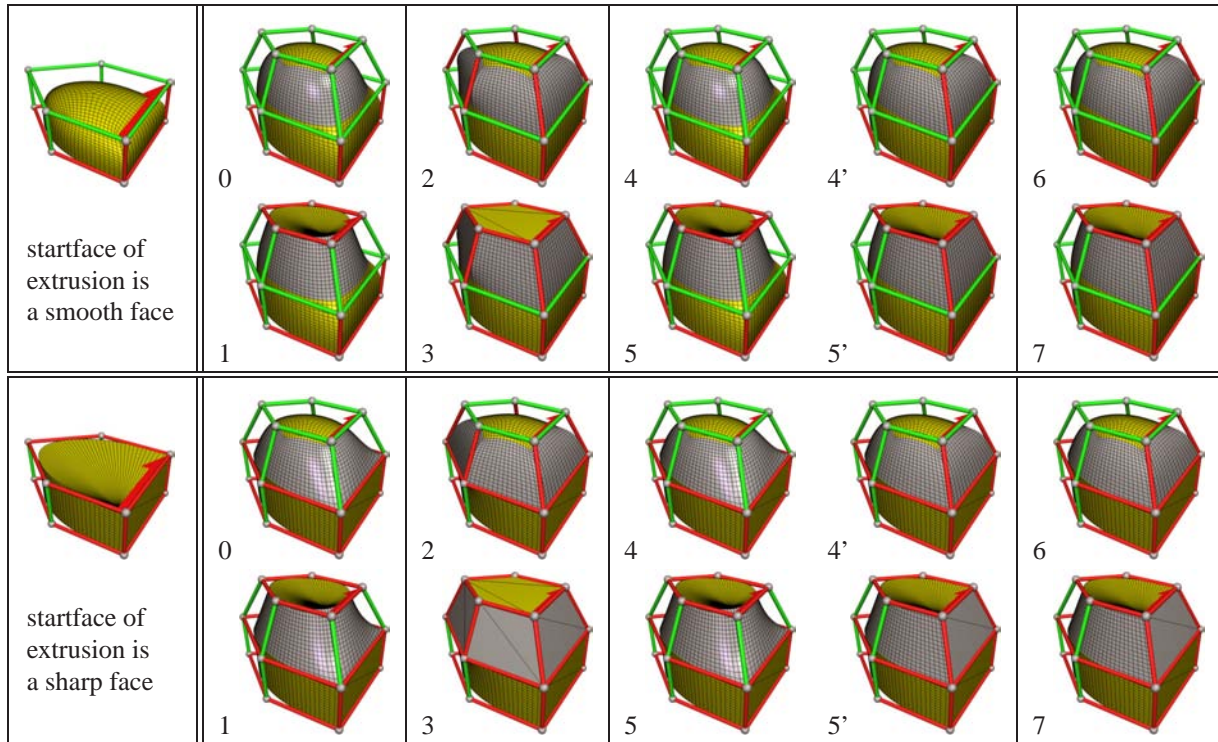


Figure 4.61: Sharpness modes for extrusion.

parallel to the original edges. But offset edges lie some distance d apart from the original edges, and this distance is the same for all edges. Offset edges lie to the left (inwards, i.e. shrinking) if $d > 0$ and to the right (expanding) if $d < 0$. The vertex positions may be computed as the intersections of consecutive displaced edge segments.

Figs. 4.59 (a) and (b) show how the position of an offset vertex is computed. Let p_0, p_1, p_2 be three consecutive vertices on the face boundary. Then the vectors $n \times (p_1 - p_0)$ and $n \times (p_2 - p_1)$ point inwards in the positive offset directions (left of boundary). These vectors are normalized to obtain two unit displacement vectors d_{01} and d_{12} . Their sum $d_{012} = d_{01} + d_{12}$ is the angular bisector of the angle between them. So the offset vertex of p_1 has to lie in direction d_{012} from p_1 . But in which distance? The scale factor w must be chosen such that $w \cdot d_{012} - d \cdot d_{01}$ is orthogonal to d_{01} .

$$\langle d_{01}, w \cdot d_{012} - d \cdot d_{01} \rangle = 0 \iff w \langle d_{01}, d_{012} \rangle = d \langle d_{01}, d_{01} \rangle \iff w = d \cdot \frac{\langle d_{01}, d_{01} \rangle}{\langle d_{01}, d_{012} \rangle} = \frac{d}{\langle d_{01}, d_{012} \rangle}$$

The crucial point with this formula is that, unlike the line intersection, it is computationally stable also in the special case when the two polygon segments are more or less collinear. This happens frequently, for example the polygon from Fig. 4.60 contains a point with collinear segments near the upper right. This diagram also shows that scaling keeps the relative lengths of the line segment constant, whereas the offset operation just faithfully ‘thickens the boundary’.

More flexibility with temporary vertex flags. Just like with the previous tools, it is possible to use the temporarily set vertex type to derive the edge sharpness assignments. An extrusion creates two kinds of edges, the boundary edges of the displaced face, which may be called *horizontal* edges, and for each vertex one *vertical* edge. The vertex type may or may not be used to derive the sharpness of these vertical edges, according to the following table. In any case, bit 0 of sharpmode specifies the sharpness of the horizontal edges, i.e., the edges of the displaced face.

- sharpmode = 0 or 1 “smooth” All vertical edges are smooth
- sharpmode = 2 or 3 “sharp” All vertical edges are sharp
- sharpmode = 4 or 5 “like vertex” A vertical edge is smooth if and only if the vertex type is ‘smooth vertex’
- sharpmode = 6 or 7 “continue” Vertical edge sharpness equal to sharpness of last corresponding vertical edge

The different possible combinations are illustrated in Fig. 4.61. The continuation modes 6/7 are somewhat special. They are supposed take into account the sharpness of vertical edges created by previous extrusions. Considering the code in Fig. 4.56, the vertical edges are created as dangling edges using `makeEV(e,e,sharp,p)`. With modes 6/7 the edge

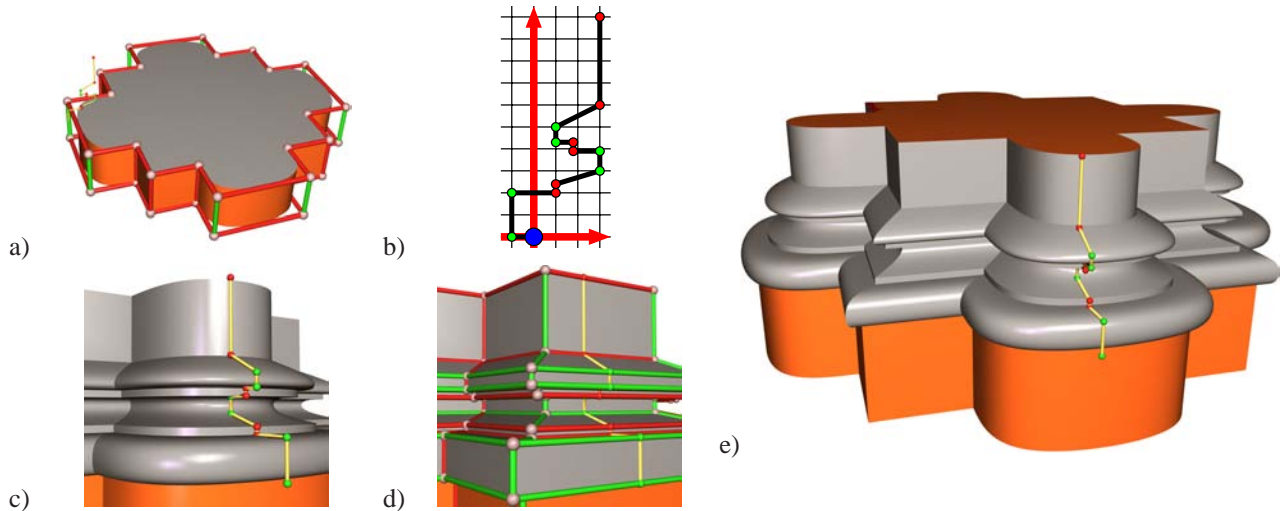


Figure 4.62: Column basis created by multiple extrusion from basis (a) with profile (b).

$e \rightarrow \text{mate}() \rightarrow \text{faceCCW}()$ is assumed to be created by the previous extrusion, and the sharpness of this edge gives the sharpness of the new vertical edge.

In case temporary vertex flags are set appropriately, the modes 4/5 yield the identical configurations as modes 6/7, which is demonstrated in columns 4'/5' in Fig. 4.61 (left side). To the right, the result of modes 4/5 is shown in the absence of temporary vertex flags, for instance when the two vertical edges below the start face have just been made sharp explicitly using `edge->sharp=true`. In this case modes 4/5 are identical to modes 0/1, and the continuation modes 6/7 should be used instead. The continuation modes, on the other hand, are not very handy when an extrusion is applied either to a face whose neighbours are not all (side-)quads, or to a double-sided face. This is the reason why both types of modes are necessary.

Multiple extrusions. So some effort was taken to assert a reasonable behaviour of the extrude operation with respect to a previous extrusion. This effort is justified by the importance of *multiple extrusions*. Multiple extrusions can be regarded as an important special case of *sweeping*, a very common operation in modeling. Sweeping means to create a surface from two curves, a *profile* curve and a *spine* curve. The surface is created by moving the profile along the spine, possibly at the same time scaling or rotating the profile. Since for extrusions the face plane of the extruded face is parallel to the plane of the original face, multiple extrusions are much like sweeping with a straight spine.

So far, every single extrusion has to be specified with three parameters, two floating point parameters and a mode flag:

- h , the amount of horizontal displacement of each edge *within* the face plane for shrinking or expanding,
- d , the amount of vertical displacement which determines how far the face is lifted up, and
- m , one of the edge sharpness modes 0-7 that were illustrated in Fig. 4.61.

These three parameters can be combined into a single 3D vector (h, d, m) of type `Vec3f`. Although it may seem strange to put the integer sharpness mode into the z -coordinate of a 3D vector, the great advantage is that a single parameter is sufficient to specify one extrusion. Multiple extrusions then correspond to an array of 3D points, which are essentially a profile curve. The modeling tool for multiple extrusions is a function declared as follows:

```
bool extrudeMulti(BRepProgressive& pcrep, Edge* edge, const Vec3f* profileBegin, const Vec3f* profileEnd);
```

A demonstration of the power of this simple function is shown in Fig. 4.62. The basis (a) was created by concatenating four arrays for the four sides of the column, each array containing six points. The points in the (sharp) corners appear twice in the concatenated array so that the `poly2doubleface` operator with mode 5 can set the temporary vertex flags appropriately. One mode 5 extrusion creates the basis which accordingly contains just four vertical sharp edges.

Then the profile curve (b) is applied using the `extrudeMulti` tool declared above. The x -axis stands for the horizontal and the y -axis for the vertical displacement; so the large blue dot in the origin is the start point (0-displacement). The polygon dots stand for the extrusion mode, which is either continuation mode 6 (green, horizontal edges smooth) or 7 (red, horizontal edges sharp). They correspond to roundings in the profile, as can clearly be seen in (c), and simply indicate the smoothness of the respective edges from the control mesh (d).

To finally create the classical column basis (e) is just a matter of calling `extrudeMulti` with the 12 profile CVs from (b): $(-1,0,6)$, $(-1,2,6)$, $(1,2,7)$, $(1,2,4,7)$, $(3,3,6)$, $(3,3,9,6)$, $(1,8,3,9,7)$, $(1,8,4,3,7)$, $(1,4,3,6)$, $(1,5,6)$, $(3,6,7)$, $(3,10,7)$

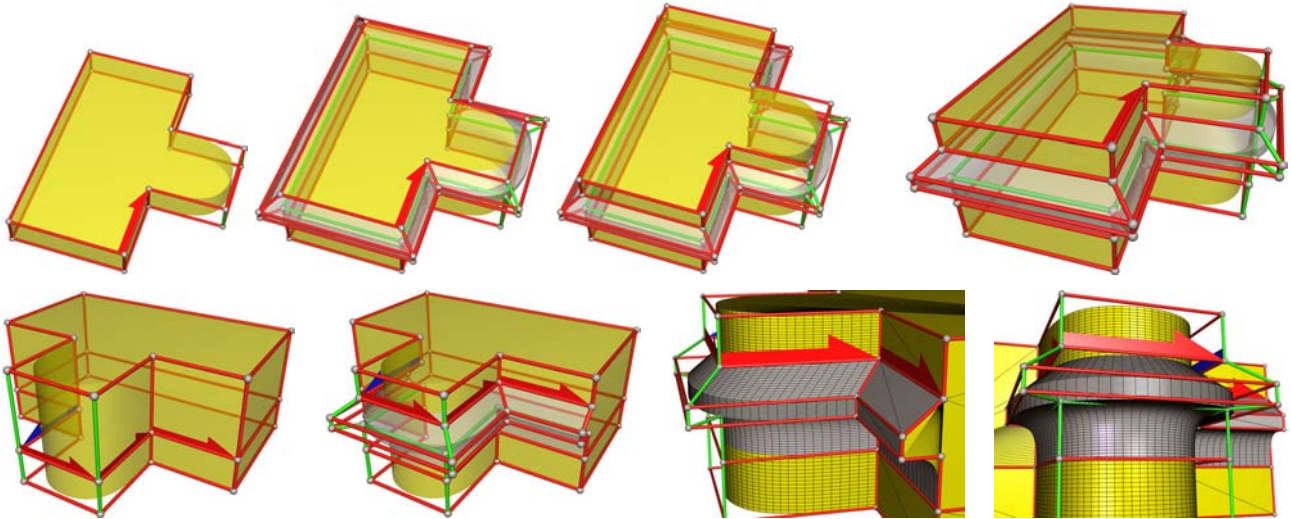


Figure 4.63: The principle of path extrusion. Standard extrusion creates stacks of slices, whereas path extrusion follows a sequence of halfedges, a *halfedge path*. Profiles may also be applied only partially around the object.

4.5.4 Path Extrusion

The extrude operator from the last section operates on a single face in the direction of the face normal. It permits not only to lift the (copied) face but also to shrink or expand it during extrusion. This is very much like a sweeping operation with a spine that is restricted to be straight vertical. More general tools are possible with additional parameters for instance to rotate the extruded face, either around the face normal, or around an axis within the face plane, or both.

But the face extrusion operator has one fundamental restriction, the construction order: Extrusions are stacked on each other and must be executed one after another, which results in a strictly linear construction sequence.

Fig. 4.63 shows an example in (1a-1d): First the base is created, then the profile, and then the upper part. Such a strict linear order does not permit to insert a profile slice ‘in the middle’, in a situation like in (2a): The profile slice was forgotten and it must be inserted subsequently. – Another problematic case is a profile that is supposed to reach only partially around. This is relevant, e.g., for creating columns that do not stand freely, but are attached to a wall.

The solution is *path extrusion*. It operates along a connected path of halfedges. Path extrusion creates a displaced copy of the edges from the original path; the latter are also called the *horizontal* edges. As before, this creates quadrangles, one for each edge along the path. A duplicated edge is always parallel to the respective original edge. It can be translated in two independent directions (i.e., in the plane normal to the edge), as shown in Fig. 4.64: Within the face plane (red), and normal to the face plane (green). So as before, an extrusion can be specified with a 3D vector (h, d, m) , where

- h is the amount of horizontal displacement of a copied edge *within* the face plane (red, x -coordinate),
- d is the amount of vertical displacement, normal to the face plane (green, y -coordinate), and, as usual,
- m is the edge sharpness mode.

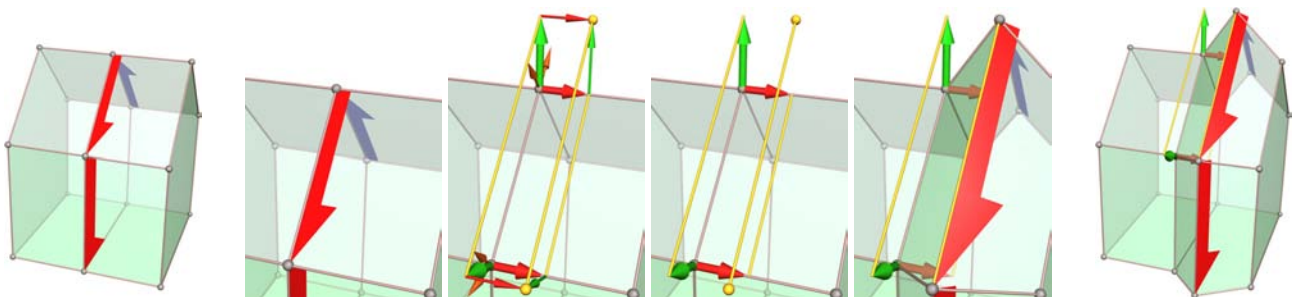


Figure 4.64: Parameters of path extrusion. For an (x, y) extrusion, the x -coordinate is the displacement within the face plane (red arrow along vertical edge), and y is normal to it (green arrow in bisector plane, also see Fig. 4.59).

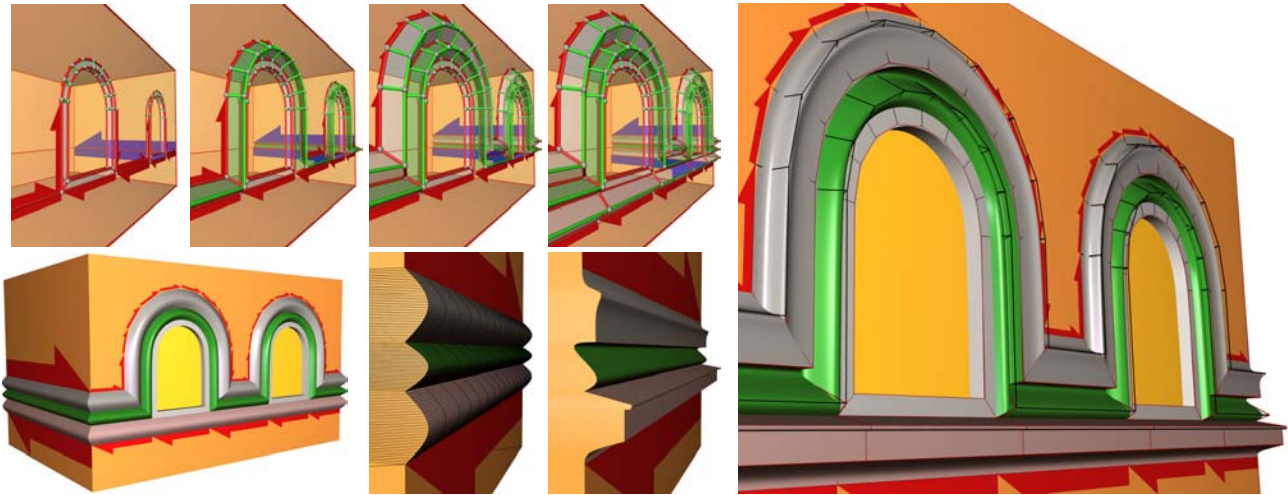


Figure 4.65: Typical façade profiles created with path extrusion. Same paths can be used with different profiles.

Note that despite the explanation of the parameters is identical to those of face extrusion (see section 4.5.3), the notions of ‘horizontal’ and ‘vertical’ are exchanged: In order to obtain the same profile with multiple path extrusions as it was obtained with multiple face extrusions, the h and d components of the profile need to be swapped.

An example of a path with four halfedges is shown in Fig. 4.63, (2a). Path extrusion yields the same profile as before in (1a-1c), but it is applied only in the front of the object (2b). The detail close-up images (2c) and (2d) show the behaviour at the end and the beginning of the path, respectively: The profile vertices are made to lie in the face plane of the respective neighbour faces (if possible). Also the profile runs only partially around the object, which is not possible with face extrusion.

Path Consistency conditions. First of all, the halfedge path needs to be singly connected, i.e., for every pair of successive halfedges e_i, e_{i+1} in the sequence, $e_i \rightarrow \text{mate}$ and e_{i+1} must share the same vertex. But not every connected halfedge path is accepted by path extrusion. An additional requirement is that there may be at most one “vertical” edge between successive halfedges. For e_i, e_{i+1} this means that either $e_i \rightarrow \text{faceCCW} = e_{i+1}$ (no vertical edge, same faces), or $e_i \rightarrow \text{faceCCW} \rightarrow \text{vertexCW} = e_{i+1}$ (one vertical edge). An example of a vertical edge is shown in Fig. 4.64 (red arrow).

In case of a vertical edge, there is still another constraint, the reflection property (Fig. 4.66). It poses a geometric restriction on the angle the path edges may have with the vertical edge between them: Both angles must either be equal or sum to 180 degrees.

Multiple vertical edges are not allowed in order to avoid too complicated configurations. With a double vertical edge for instance $e_i \rightarrow \text{faceCCW} \rightarrow \text{vertexCW} \rightarrow \text{vertexCW} = e_{i+1}$ for one i . Both $e_i \rightarrow \text{faceCCW}$ and $e_i \rightarrow \text{faceCCW} \rightarrow \text{vertexCW}$ are vertical edges where displaced vertices must be inserted. They must be connected by another edge, which corresponds to no edge on the path, so it is not an offset edge; and instead of a quadrangle, it creates a triangle. Furthermore, at all inserted vertices the reflection property must hold, which requires a *very* special geometric edge configuration.

It may seem that profile extrusion is not very useful since so many constraints need to be met. But all on the contrary, it is a versatile tool for the domain of architecture; for instance the decoration above the window in Fig. 4.52 was created with it. A path on a surface is a very natural concept, and the separation between path and profile permits much flexibility. The example in Fig. 4.65 demonstrates switching of profiles, and it uses a path without vertical edges around the windows.

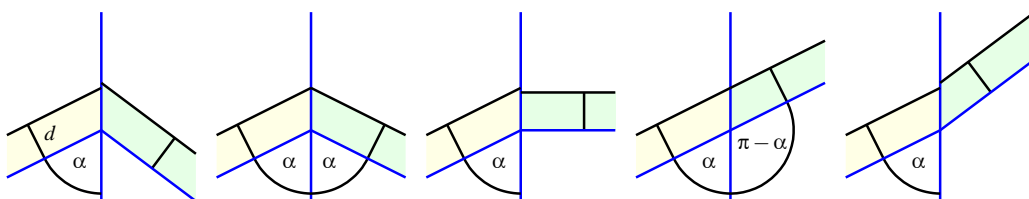


Figure 4.66: Reflection property. The offset edges in fixed distance d only meet in one point if both angles are equal, or if they sum to 180 degrees. Note that this holds also if the left and right surface normals are different.

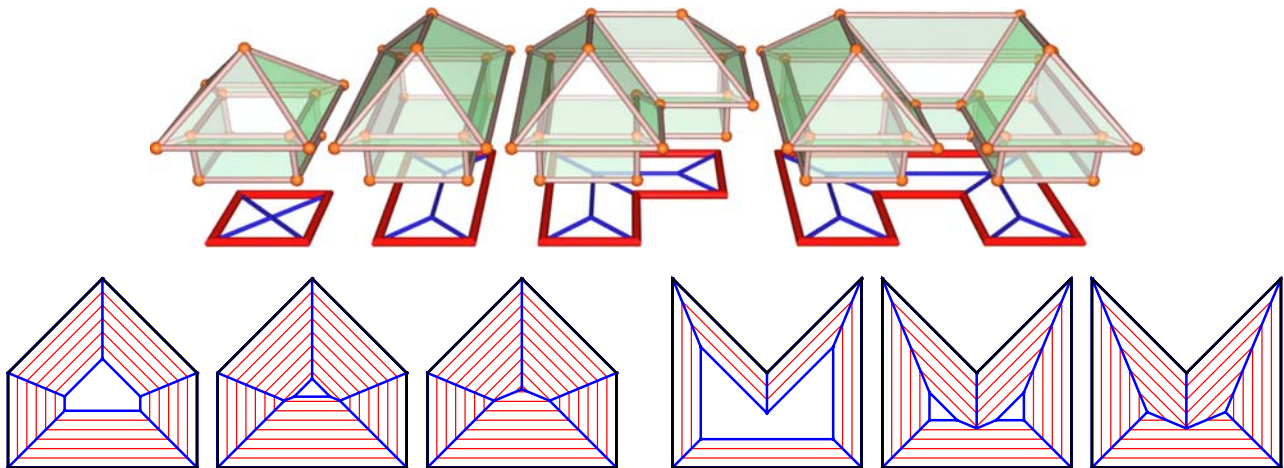


Figure 4.67: The straight skeleton. When building roofs, simple extrusion leads to self-intersections. It is very instructive to consider the way carpenters build a roof. Lower row: Two types of problems occur when the horizontal offsets become larger, *collapsing edges* (a-c) and *intersecting angular bisectors of reflex vertices* (d-f).

4.5.5 Intersection-free Extrusion and the Straight Skeleton

One particular problem for any kind of extrusion that includes offset operations, such as to shrink or expand a contour, is the possibility that self-intersections can occur. The amount of offset that is legal and does not lead to self-intersections is often not known beforehand. Recall the example from Fig. 4.60 (b), where the offset contours from opposite sides of the non-convex polygon nearly touch at its narrowest waist. It is very tedious to determine the interval of legal offset distances by trial and error; and if the offset operation is applied to a polygon that is dynamically generated, for instance as a part of a complex construction, self-intersections must be reliably repaired automatically, rather than only avoided.

Reasons for self-intersections. Analyzing the problem one finds that two types of situations lead to self-intersections. They are shown in Fig. 4.67, lower row. The first three pictures (2a-c) illustrate the problem of collapsing edges: The offset copies of the left and right edges get shorter and shorter, until they collapse to a single point. The second type of problem arises when an offset vertex approaches an offset edge (or vertex) on the opposite side, as shown in the three pictures to the right, Fig. 4.67 (2d-f). Such a vertex is called a *reflex vertex*: The interior polygon angle is more than 180 degrees. Traveling CCW along the polygon boundary one has to turn right at a reflex vertex, rather than left as usual. Every non-convex polygon has at least one reflex vertex, and vice versa.

Fix self-intersections from collapsing edges. Both situations can be resolved in basically the same way. The idea is to imagine a carpenter building a roof. It is economic to build up a roof of planar parts, each part attached to one wall. It is also reasonable for all planar roof parts to mount with the identical slope. Each of the walls corresponds to one edge of the ground polygon of the house. So to plan the roof is basically a 2D problem: Every polygon edge is offset inwards to determine the position of the next roof lath; in reality the individual roof tiles are attached to these laths. For consecutive polygon edges, the laths meet in the *angular bisector* of the interior angle at the common vertex. A single angular bisector ray, also just called the *bisector*, emanates from every vertex of the polygon. This property holds also when an edge collapses to a single vertex: The left and right neighbour faces now become neighbours, and their line of contact is also the angular bisector of their respective base edges. This is shown in 4.67 (b). When all remaining offset edges eventually collapse to a point or a line, as in 4.67 (c) and (f), the roof plan is finished. This plan is called the *straight skeleton*.

Fix reflex rays that hit opposite edges. The bisector from a reflex vertex can split an opposite edge in two, as in 4.67 (e,f). A new vertex is inserted at the end of the bisector. Note that this vertex has the same distance from three lines in 2D: From the line through the opposite polygon edge, and from the lines through the two edges that meet at the reflex vertex. Just as in the case of a collapsing edge, faces become neighbours that were not neighbours before. Two new bisectors emanate from the new vertex, computed from the relative orientation of the respective original polygon edges.

Note that many different special cases are possible: It may be that several edges collapse at the same time, for instance when vertices lie on a circle. The bisector of a reflex vertex may meet with another reflex bisector from the opposite side, or with a collapsing edge. But they may also just fail to meet exactly: The *straight skeleton* is quite unstable sometimes.

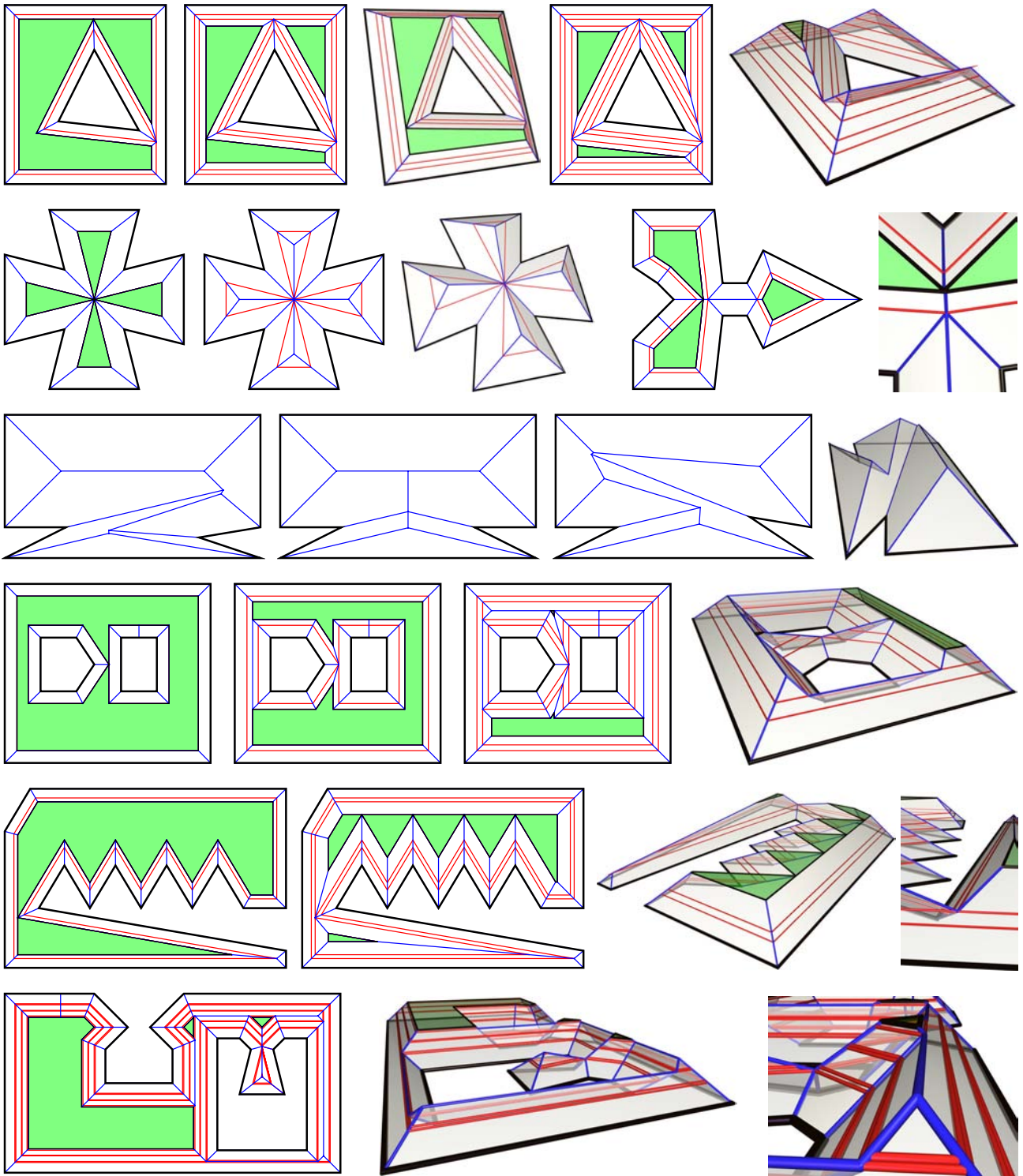


Figure 4.68: Examples of the straight skeleton. Every red line corresponds to one or more *edge*, *split*, or *vertex* events. Rings contain mostly reflex vertices (1a-e). Whenever a reflex bisector hits an offset edge (split event), it splits the polygon in two (1a,b). The new vertex belongs to both, with two new bisector rays in either direction (1d). Reflex bisectors may hit not only edges but also other reflex bisectors (2a-c) in a *vertex event*. But they may also just fail to do so (2e): One of the two edges in (2d) is slightly perturbed, and the vertex event turns into a series of edge events. The straight skeleton is an unstable problem, since small variations of the vertex positions can sometimes yield greatly different skeletons (3a-d). Approaching parallel lines collapse into a line rather than a point, with several split events (4b) and edge events happening at the same time (4c). With coincident events, the correct geometric and temporal order is crucial (Rows 5, 6).

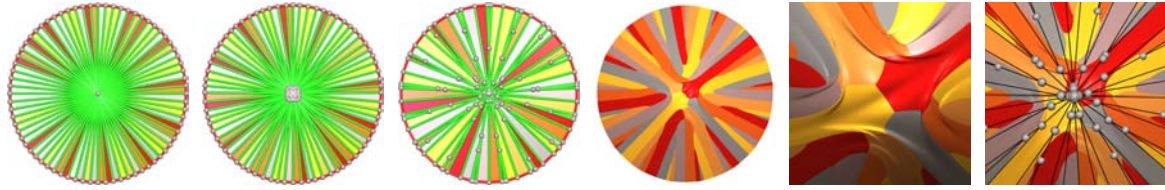


Figure 4.69: Straight skeleton of points on a circle. The angle between 72 consecutive points on the unit circle was randomly distorted by 0.5 (b) and 3.5 degrees (c) to simulate the effect of an uneven sampling of a smooth curve. The straight skeleton is very sensitive to this sampling noise. The resulting subdivision surface (d) exhibits wobbling (e) since the control polygon has many very short edges near the circle center (f).

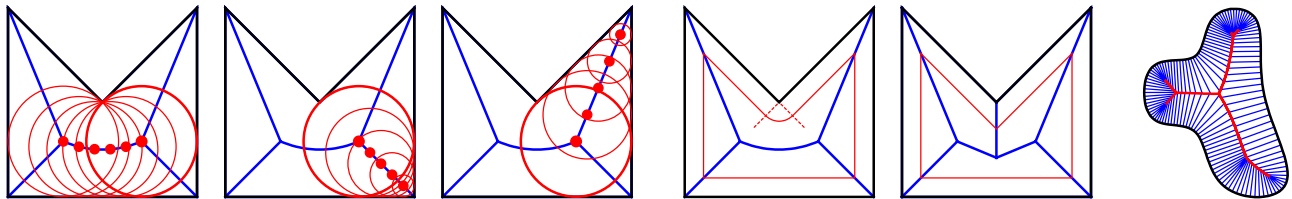


Figure 4.70: The *medial axis* of a polygon is the set of the centers of all maximal circles lying inside the polygon (a)-(c). For reflex vertices, the medial axis contains also curved (parabolic) parts (a). The medial axis can also be obtained from an offset operation. Considering the polygon boundary as point set, the Euclidean distance leads to an offset curve that, unlike the straight skeleton (e), contains also curved (circular) segments for reflex vertices (d).

The straight skeleton and the medial axis. In the computational geometry literature the straight skeleton was first introduced by Aichholzer and Aurenhammer in 1995 in [AAAG95]. They observed that the construction of a roof was similar, but not identical to the *medial axis* of a polygon. The latter is a well known concept in computational geometry with a huge number of applications. The medial axis of a polygon is defined as the set of points with more than one closest point on the polygon boundary. In case the polygon is convex, it is a network of straight line segments. But in case the polygon contains reflex vertices, the medial axis contains also curved parabolic segments, as shown in Fig. 4.70.

The reason is that the locus of the points that have the same distance from a given point and a given line is a parabola. This is easy to see, e.g., considering the set of points $(x, y) \in \mathbb{R}^2$ that have the same distance from the x -axis and the point $(1, 0)$. The length of the difference vector $(x - 0, y - 1)$ must be identical to the height y . This yields a parabola:

$$y = \sqrt{(x-0)^2 + (y-1)^2} \Leftrightarrow y^2 = x^2 + y^2 - 2y + 1 \Leftrightarrow y = \frac{1}{2}(x^2 + 1)$$

The medial axis can also be characterized as a part of the Voronoi diagram of the polygon vertices and edges. Another way to construct it is by an offset operation, where the polygon is used as the path of a circular pen (Fig. 4.70 d). The breakpoints between consecutive line segments and radial arcs trace out the medial axis [EE99]. But the resulting offset curve contains also curved (circle) segments for each reflex vertex, which is not desirable. The solution is the straight skeleton: It considers the distance from the *line* through a polygon edge, rather than from the polygon edge as a point set. The straight polygon itself as well as all offset polygons contain only straight line segments (Fig. 4.70 e), hence the name. As observed by Eppstein and Erickson in [EE99], the difference between medial axis and straight skeleton can also be characterized by the distinction between line joints in vector drawing programs, the rounded and the *mitered* joint styles.

Literature on the straight skeleton. It may be surprising that the medial axis can be computed in linear time, which was shown by Chin et al. in [CSW95]. But unfortunately, equally efficient algorithms for the straight skeleton are not known. The original straight skeleton algorithm from Aichholzer and Aurenhammer in [AAAG95, AA96] has time complexity $O(n^2 \log n)$ for a polygon with n vertices. It can be improved with a quadtree to achieve a quadratic complexity of $O(nr + n \log n) = O(n^2)$, where $r < n$ is the number of reflex vertices, as it was reported by Eppstein and Erickson in their beautiful article on “*Raising Roofs, Crashing Cycles, and Playing Pool*” [EE99]. In this article they propose another, more involved, ‘reflex-sensitive’ approach for computing the straight skeleton that runs in sub-quadratic time $O(n^{1+\epsilon} + n^{8/11+\epsilon} r^{9/11+\epsilon}) = O(n^{17/11+\epsilon})$. They claim that a ‘practical variant’ of their algorithm runs in $O(n \log n + nr)$ time with $O(n + r^2)$ space. This bound is improved by Cheng and Vigneron in [CV02] who propose a randomized algorithm that runs in $O(n \log^2 n + r^{17/11+\epsilon})$ for degenerate polygons (with vertex events) and in $O(n \log^2 n + r\sqrt{r} \log r)$ otherwise. Their algorithm cannot handle polygons with holes, though.


```

INTERSECTIONFREEOFFSET(Polygon P, d)
1  for all vertices:           compute angular bisector and identify reflex vertices
2  for all polygon segments:  compute edge events
3  for all reflex vertices:   compute split events and vertex events
4  Create event queue Q from all events with event time  $t < d$ , sorted by t
5  while Q not empty process next event according to event type
6      do remove or split boundary segments of the offset polygon
7          create the edges, vertices, and faces of the straight skeleton
8          update Q according to the changes made to the polygon
9  for all vertices:           create dangling vertices on bisector rays
10 for all polygon segments:  add 'horizontal' edges at distance d

```

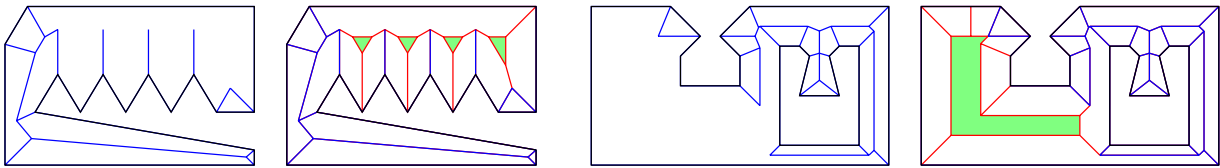


Figure 4.71: Intersection-free offset polygon at distance $d > 0$. First all events that occur in the interval $[0, d]$ are executed (a,c). Then the vertical and horizontal edges are inserted (red). The horizontal edges form the boundaries of the actual offset polygons, shown in green (b,d). These boundaries are in distance d from the original polygon.

Computation of the straight skeleton. The offset distance at which a polygon segment collapses can be computed in constant time. It is the distance between the 2D line along the segment and the point c that is the intersection of the two bisector rays from its end vertices. In case both of them are reflex vertices, the intersection point lies on the exterior side, and the segment's edge event time is negative; it may as well be infinite in the special case of parallel bisector rays.

The effect of the reflex bisectors is more difficult to compute since by its nature it is non-local. Note that the reflex bisector for unit edge speed can become arbitrarily long if the interior angle approaches 360 degrees. One source of difficulties are multiple reflex bisectors that meet in a single point, as in the center of the cross in Fig. 4.68 (2a)-(2c). This case is treated separately as a new event type by Eppstein et al, who call it a *vertex event*. The reason is that unlike edge or split events, a vertex event can introduce a new reflex vertex into the shrinking polygon. The failure to process a vertex event appropriately can have a drastic effect on the straight skeleton, as shown in Fig. 4.68 (3a)-(3d): A vertex event is a very special case that can be thought of as being surrounded by split events on either side. In any case very small variations of the input polygon may drastically change the result.

Both with split and vertex events the total number of reflex vertices decreases, since also each vertex event removes (at least) two reflex vertices. Vertex events are a special case because they do not occur when the vertices are in general position. So polygons whose straight skeletons have vertex events are considered degenerate by most authors. It appears however that they are quite important in practice, e.g., for constructing roofs.

Intersection-free offsets/extrusions and the straight skeleton. A *straight skeleton algorithm* takes basically the form of a sweepline algorithm, similar to the triangulation algorithm in section 4.2.2. But instead of a single sweep line, the sweep parameter measures the offset distance from the polygon boundary. This offset distance can be seen as a time parameter (of the sweeping motion), as well as the increasing height of the roof. Three types of events are processed:

- **edge event:** a polygon segments vanishes because the bisectors of its two end vertices cross, see Fig. 4.67 (2b)
- **split event:** a polygon segment is split because it is hit by the bisector of a reflex vertex, see Fig. 4.67 (2e)
- **vertex event:** two or more reflex bisectors meet in a single point, see Fig. 4.68 (2a)

It must also be noted when an event makes a sub-polygon completely vanish, e.g., when all remaining edges collapse at the same time (Fig. 4.67 (1a)), or when only 2-gons remain (Fig. 4.67 (1b-d)). The basic outline of a straight skeleton algorithm is given in Fig. 4.71. The computationally most expensive part is to find out which polygon segment is hit by a reflex bisector. A priori this can be any other segment, so that step 3 has complexity $O(m)$ when all segments need to be tested with each reflex bisector. This algorithm can be used to compute

- **the straight skeleton:** The offset $d = d_{\max}$ is so large that no more events happen after it
- **an intersection-free offset:** Part of the 2D straight skeleton and the offset polygon at distance $d < d_{\max}$
- **an intersection-free extrusion:** In addition to the offset distance d the elevation h is specified (roof slope d/h)

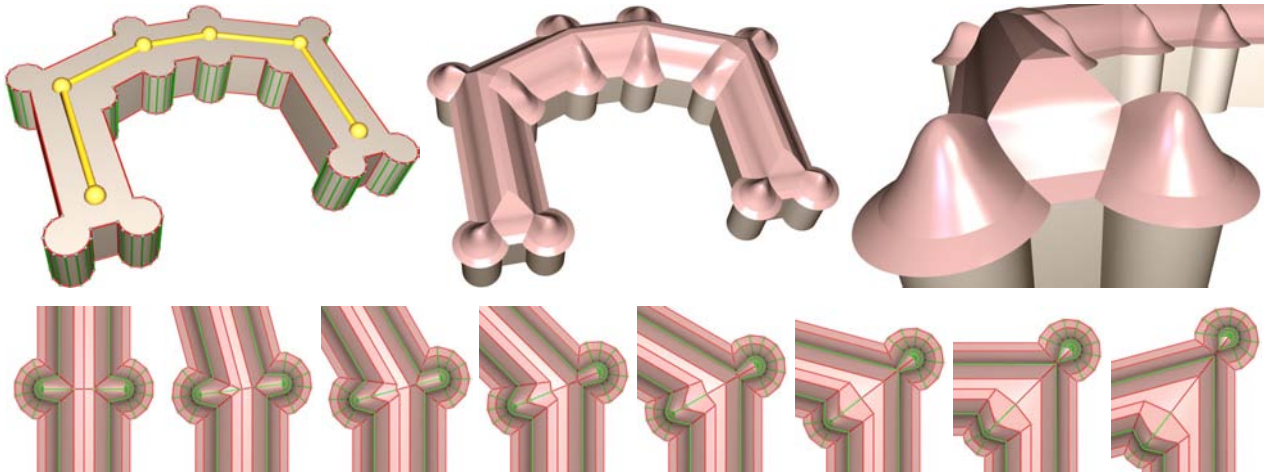


Figure 4.72: The roof of a castle. The roof would be very hard to realize without an automatic straight skeleton computation, due to the many possible control mesh configurations from different angles between the aisles.

Building roofs and vertex events. Especially for building roofs, the degenerate polygons where the straight skeleton exhibits vertex events are indispensable. Carpenters actually favorize corner configurations like in Fig. 4.67 (1c) and (1d), because they keep the shape of the individual roof parts simple. Also reflex vertices are not always a special case. First, it is sometimes desirable to compute the straight skeleton in outward rather than inward direction, i.e., to inflate rather than to shrink the polygon. In this case all normal (convex) vertices become reflex vertices and vice versa. Second, if a polygon has rings, normal ring vertices are also reflex vertices for inward offsets, as in rows 1, 4, and 6 of Fig. 4.68.

A refined example of a roof that is exclusively built with intersection-free extrusion is shown in Fig. 4.72. It is made by multiple extrusions with different slopes, and after some of them the horizontal edges of the extruded faces were made smooth. The second row of images shows the power of this approach, since the intersection-free extrusion permits to generate the roof automatically despite the problematic ground polygon that leads to coincident edge collapse events of a high degree at the circle centers, as well as to many vertex events.

Straight skeleton of polygons sampled from a smooth curve. Every edge event collapses an edge, removes (at least) one boundary component, and thus, destroys information. It is interesting to examine the straight skeleton from polygons that approximate a smooth curve by discrete samples. The example in Fig. 4.73 shows the straight skeleton of the polygon from Fig. 4.60, which was generated from a B-spline curve. With increasing sampling density the angular bisectors converge to the 2D normal vectors of the polygon boundary. But a smooth curvature variation implies that nearby points have similar osculatory circles, and the respective normal vectors point to the centers of these circles. In case of a convex, smoothly rounded curve all normal vectors nearly intersect in a common point, the approximate midpoint of the osculatory circles. In case the rounding is not a perfect circle, the midpoint degenerates to a curve. Smooth elongated shapes therefore exhibit the phenomenon of a ‘spine’, which may also bifurcate. It is plausible that the medial axis and the straight skeleton converge to the same limit spine, e.g., the red spine shown in Fig. 4.70 (e): Note that in case of a convex polygon, the straight skeleton is identical to the medial axis.

One especially interesting smooth curve is the circle. The experiment in Fig. 4.75 (1c) and (1d) shows a parabola that appears as the projection of the intersection of two surfaces: (i) the offset surface swept from a half circle, and (ii) the offset surface swept from a line. So in this case also the straight skeleton produces (an approximation to) a parabola. This shows again the tight relation between the medial axis and the straight skeleton.

The straight skeleton, however, is extremely sensitive to sampling noise. This is illustrated with the experiment in Fig. 4.69. It shows in (a) the projection of the straight skeleton of a regular 72-gon: a perfect polyhedral cone where all bisectors meet in the center in one large edge event. But then jitter is introduced, with the vertices sliding along the circle, but keeping the same distance to the center. With increasing jitter, the bisectors more (b) and more (c) deviate from the true circle normals. As a consequence, the intersection points are more and more spread as well.

The subdivision surface from a straight skeleton. Applying the intersection-free offset to a polygon sampled from a smooth curve immediately raises the question whether the skeleton can also be used to generate a smooth surface. The straight skeleton from the jittered 72-gon on the unit circle exhibits very short edges near the center, shown in Fig. 4.69 (d,f). They attract the surface unproportionally strong, which leads to wobbles that may result in surface folding (e).

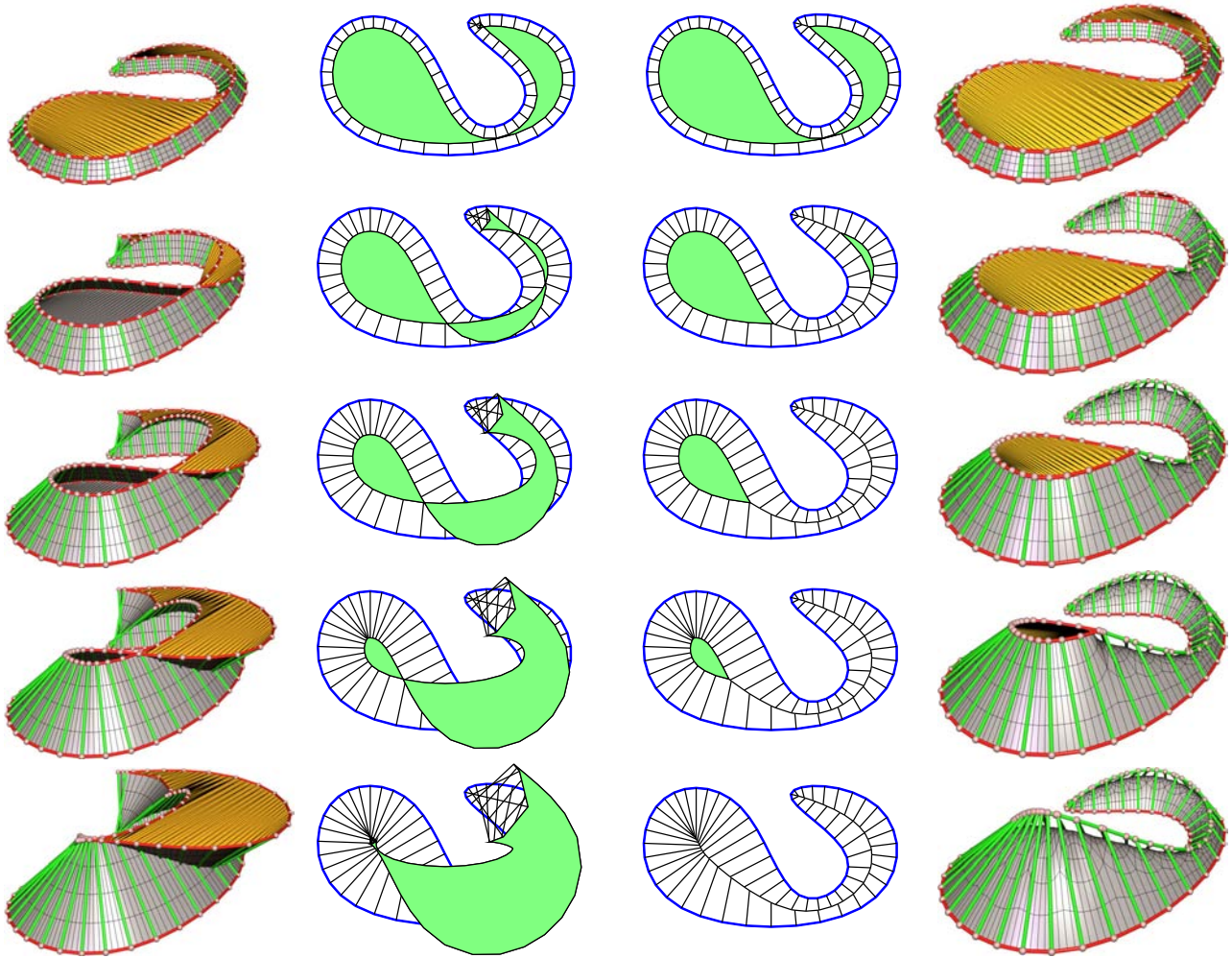


Figure 4.73: Comparison between polygonal offset and straight skeleton of the sampled curve from Fig. 4.60. The straight skeleton forms a spine curve (5c). It appears gradually, beginning with a reflex part from the non-convex part in the middle (1c) and the high-curvature part at the right end (2c). The high curvature leads to a ‘focal self-intersection’ artifact (5b) from bisectors in the right and left shape ends that (almost) meet in single points.



Figure 4.74: Subdivision surface from irregular straight skeleton. The small edges in the center of the left end lead to surface wobbling (1a, 1b, 2a), an effect that is explained in Fig. 4.69. The surface on the thin end is not completely smooth either because of the varying surface degrees. Also note that the spine curve contains several slight hills (1c,2c). The reason is that more bisector rays from the outer than from the inner part of the polygon boundary arrive at the spine. So the rays from the outer boundary are slightly longer, i.e., they reach higher.

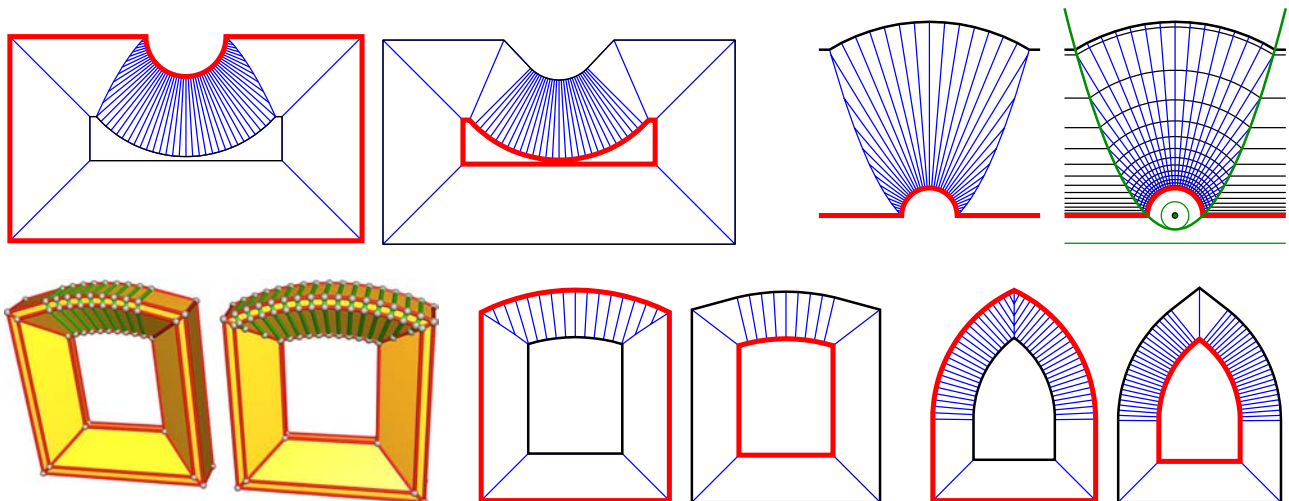


Figure 4.75: The offset operation is not always invertible since every edge collapse event removes information. A shape (non-convex red contour) is first offset inwards (1a,2c,2e) and then outwards (1b,2d,2f). If any edges were collapsed during the first offset, the reversed offset does *not* reconstruct the original shape. A parabolic curve can be obtained also with the straight skeleton, with the offset of a circle segment set into a straight line (1c,d).

Better results can of course be achieved with more evenly spread curve samples. But a closer inspection of Fig. 4.73 reveals that the wobbles are an inherent problem of this method and can not always be avoided. The images (5b) and (5c) show a direct comparison between simple and intersection-free extrusion. The simple extrusion suffers of course from self-intersections. But the surface subdivision proceeds locally, and the tessellation of the subdivision surface in (5a) is much more regular than the one in (5d), which exhibits some irregularities.

They were further examined in Fig. 4.74. On the left (bigger) part of the polygon, the accumulation of the intersection points near the top of the hill leads to an effect similar to 4.69 (d): Short control mesh edges attract the surface and lead to wobbles (1b). Another source of wobbles is the uneven distribution of face degrees, the reason for which becomes evident at the right (small) end of the polygon: Although the sampling rate is very similar on both sides, more bisector rays reach the spine from the exterior (right) side; consequently the faces on the inner side have higher degrees, thus they attract the surface more.

Reversible offset operation and the need for multiple intersection-free extrusion. Every collapse event removes information, as clearly illustrated in Figs. 4.75 (1a) and (1b). This is a difference to simple extrusion and offsetting where all vertices remain in the polygon irrespective of the geometric inconsistencies they cause. One consequence is that intersection-free extrusion is not reversible, i.e., the original polygon cannot be reconstructed from the offset polygon. It is not possible to go back from (1b) to (1a) by applying an exterior offset. Considering Fig. 4.73 it may be quite interesting though to determine all polygons (or smooth curves) that result in the same spine (also see Fig. 4.70 (f)).

Note that also simple extrusion is no longer invertible as soon as one edge gets reversed. This is shown in Fig. 4.76 (c): Whenever the polygon intersects itself, e.g., it forms an ‘8’ shape, also called a bowtie, the wrong side of one edge is considered as inside, and the direction of the bisector ray suddenly flips by 90 degrees. Therefore 4.76 (b) can not be reconstructed from 4.76 (c) either, although no vertices got removed.

Also note that intersection-free shrinking *can* be reversed in case exclusively split events took place. An example are the three triangles in Fig. 4.68 (1b), from which it is possible to reconstruct the original quad with the triangular hole.

An important motivation for a reversible offset operation are multiple intersection-free extrusions. The usefulness of multiple extrusion was demonstrated, e.g., in the column from Fig. 4.62. During the vertical extrude motion, the polygon can be shrunk and expanded, which basically means to apply a vertical profile. Without multiple intersection-free extrusion, one has only two options, both of which are equally unsatisfactory:

- to limit the shrinking/expansion rate of the vertical profile so that the (horizontal) polygon experiences no self-intersections – this option was chosen in Fig. 4.62 –, or
- to use only *monotone* profiles that, unlike the profile from Fig. 4.62, either only shrink or only expand the polygon. This solution is at least acceptable for creating (most) roofs.

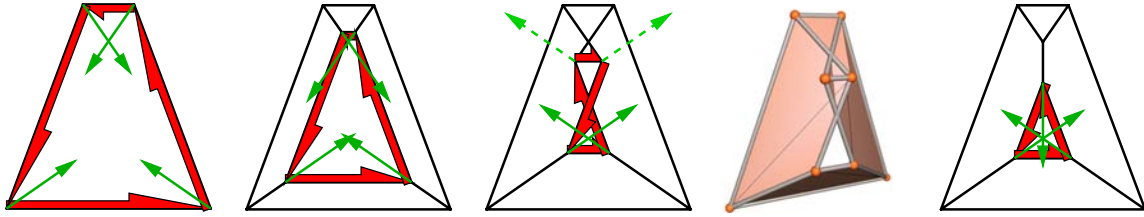


Figure 4.76: Flipping bisector angles due to bowtie offsets. Also with multiple extrusions, the bisector direction remains the same (a,b). An exception occurs only when an edge is reversed because of an unhandled edge collapse event (c),(d): The dashed bisectors in (c) now point outwards rather than inwards, and they are flipped by 90 degrees. As a consequence, the offset is no longer reversible. (e): The edge event is correctly processed.

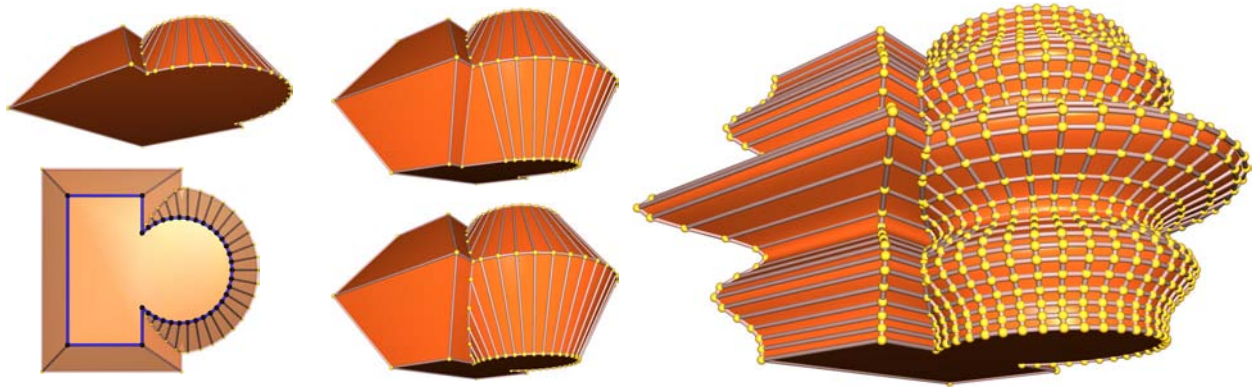


Figure 4.77: Reversible multiple offsets. Information lost with edge collapse events (1b) is re-inserted by issuing the respective inverse Euler operations (2b). This enables arbitrary stacking of multiple offset operations (right).

Multiple intersection-free extrusion. The input parameters are one (or more) planar (or co-planar) mesh faces and the vertical profile. The profile is a sequence of (x, y, s) triplets, where as before x is the horizontal offset distance for shrinking ($x > 0$) or expansion ($x < 0$), y is the height of the vertical extrusion at distance x , and s is the sharpness of the horizontal edges. The sharpness is not a mode flag but just either 0 or 1. The key to realize an ‘invertible straight skeleton’ is now to divide the original algorithm from Fig. 4.71 into two different phases. First of all the interval $[x_{\min}, x_{\max}]$ of the required expansion and shrinking of the polygon is determined, with $x_{\min} \leq 0$ and $x_{\max} \geq 0$. The two phases are the following:

1. **Compute all the events in $[x_{\min}, x_{\max}]$**
First compute only two sequences of events, namely the the collapse, split, and vertex events that occur when going from offset distance 0 to $x_{\min} \leq 0$ (expand), and, in a second pass, from 0 to $x_{\max} \geq 0$ (shrink).
2. **Process the profile**
Each item from the profile, from begin to end, is processed. This involves first to execute all the events on the way from offset distances x_{i-1} to x_i , and also in the right vertical height linearly interpolated between y_{i-1} and y_i . Second, the horizontal edges for the respective item are created, with the prescribed sharpness s , thereby inserting also the missing vertical edges.

By the separation into two phases the manipulation of the mesh is decoupled from the computation of the events. This is the key for processing arbitrarily complex vertical profiles with any combination of positive or negative vertical displacements and any expansion and shrinking – which makes it very easy to create complex models such as in Fig. 4.77 (right).

Subtleties with numerical calculations. The first phase of the algorithm, the computation of the events, uses geometric predicates that pose delicate numerical problems because many practically relevant polygons are degenerate. Numerical issues are not addressed by the cited publications; they are usually delegated to general computational geometry libraries such as CGAL [FGK*00] or LEDA [KN04]. But this is not always necessary. – Since both the extent of the polygon and the maximum extent of the profile are known, the extent of the result is bounded. This means that *fix-point arithmetic* can be used that provides much higher accuracy with the same number of bits. When the vertex coordinates are transferred to integer numbers, the slope of two lines can be compared more accurately. Then the intersection of nearly parallel lines, such as neighbouring bisector rays of a densely sampled curve, can be determined with a guaranteed tolerance.

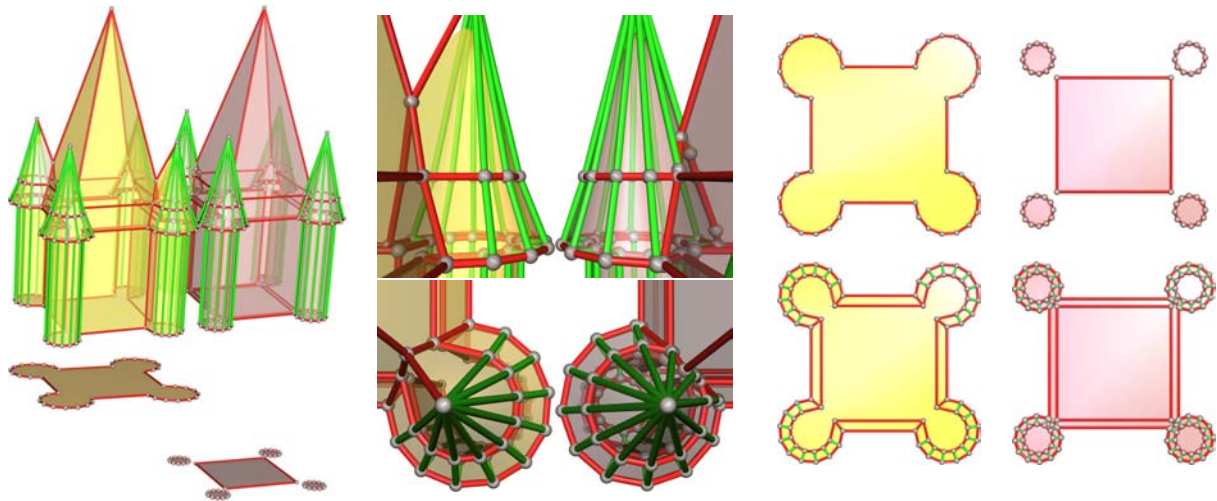


Figure 4.78: Two ways to build a tower with four corner towers. Tower A (yellow, left) is built from an intermediate level while tower B (reddish, right) is built from the ground. Polygon A (1c) has greater extent, but polygon B (1d) contains more information, the full circles of the corner towers. So the hats of tower B are more complete (2b).

Reversible intersection-free offset. The processing of every event corresponds to a short sequence of Euler operators.

An edge event leads to one new vertex connected to two (or more) bisector rays. A typical case is one (or more) triangles made of two bisector rays and a horizontal edge, such as in the left of Fig. 4.71 (c). It corresponds to one dangling makeEV followed by one or more makeEF.

A split event leads to a single, dangling, bisector edge, such as for example the three upward edges in Fig. 4.71 (a). It is created by the dangling edge version of the makeEV Euler operator (see Fig. 4.47 (1b)).

A vertex event is the special case of two or more reflex rays that meet in a single point, such as in Fig. 4.68 row 2, and (3b), as opposed to (3a) and (3c). It also corresponds to a dangling makeEV followed by one (as in 4.68 3b) or more (as in 4.68 2a) makeEF operators.

Note that in all of these cases, it may be that applying makeEF is not legal because the two vertices to be joined come from two different rings or connected components. This can happen especially when an expanding offset is applied to a polygon that is not connected, i.e., that has several connected components, as in Fig. 4.78 (1d). In such cases a slightly more complex procedure must be used to join the two vertices.

The main idea for a reversible intersection-free offset is to find Euler operations that correspond to the inverse of an event. After an edge event, for instance, a single (non-reflex) bisector ray emanates from the newly inserted vertex. So the inverse edge event contains this bisector in reversed direction, leading to the event point, from where two (or more) rays emanate in the inverse direction of the original bisectors.

In a similar fashion Euler operations can be found for inverse split and vertex events.

Results: Multiple extrusions and building church towers. The great benefit from a reversible intersection-free offset operation can be clearly seen in Fig. 4.77. Subsequent shrinking yields the original polygon only when the intersection-free offset is reversible (2b). Multiple stacking of the intersection-free offsets yields a CSG-like result, the union of a distorted cylinder and a distorted box. Both the horizontal and the vertical resolution can be arbitrarily increased without loss of information, a process that also converges towards a well defined surface in the limit.

The reversible intersection-free offset permits much more flexibility in modeling. This is also illustrated with the example of a tower with four smaller corner towers in Fig. 4.78. Without the reversible intersection-free offset, it is best to start at the level where the polygon has the greatest extent, which is the intermediate level at the base of the roof (2a). One way to build the tower is to combine four 270 degree circle segments in the corners into one polygon, which is then turned into a double-sided polygon (1c). Towards the roof, it is shrunk until it collapses at the tip of the tower. Towards the base, it is only slightly offset inwards (2c) and then vertically extruded to the ground.

But instead of starting with the largest polygon, it is better to start with the polygon that contains the most information. This is a polygon with 5 connected components, four full circles and a square, located at the ground (1d,2a). It is offset outwards such that the circles touch the square to create the actual base polygon (2d). The great advantage of using one multiple extrusion is now that the full circles re-appear when the roofs of the corner towers are created. This gives a more natural result than with the first alternative, as revealed by the direct comparison in 4.78 (1b,2b).



Figure 4.79: Ornamental profile using reversible offset. Highly non-convex shapes require a reversible straight skeleton approach. In the right column (3c-7c) the same profile is used as in Fig. 4.62.

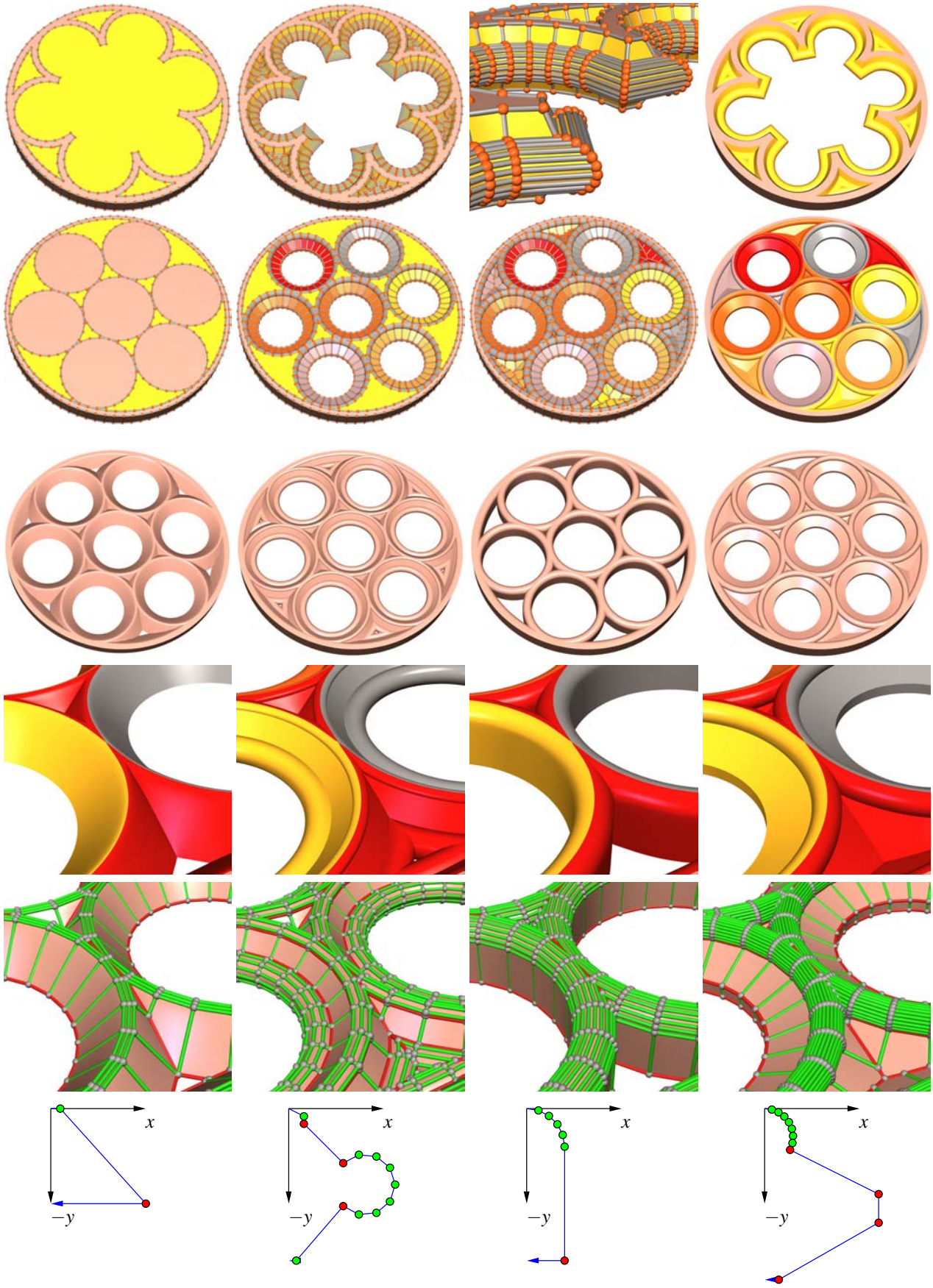


Figure 4.80: Tubular shapes through intersection-free extrusion with a half-circle profile. Unlike in Fig. 4.79 the regions are directly attached to each other: Every pair of adjacent circles share a vertex.

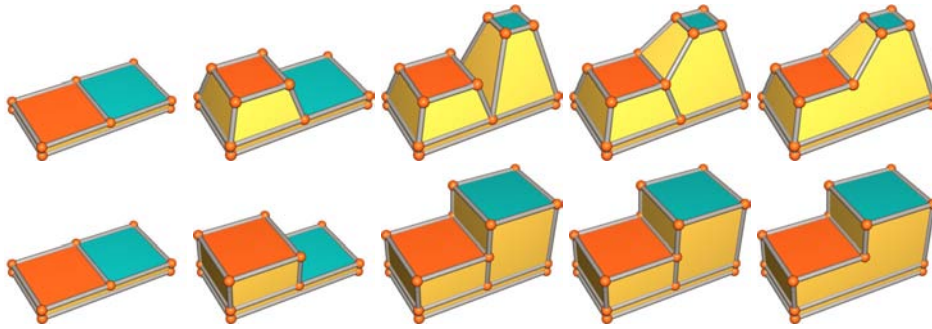


Figure 4.81: Problem with intelligent modeling tools. In case the extrusions do *not* shrink the polygon the result (2c) is perceived inconsistent because it has the same connectivity as (1c). Remedy is either (2d), which is like (1d), or (2e). The problem is that both remedies can only be applied in quite special situations that can also be hard to detect automatically. And as a consequence, the output of intelligent modeling tools may vary intransparently.

Results: Decorated circular window. This example demonstrates the full power of multiple reversible intersection-free extrusion: Intersection free extrusion permits to apply offsets to arbitrarily shaped simple polygons, and its reversible version adds the possibility to arbitrarily mix expansion and shrinking in the vertical profile. In combination, it permits to obtain impressive results such as the rosette windows from Fig. 4.79 in a very simple way.

The image 4.79 (1a) shows a large circle polygon (i.e., a regular n -gon) with four non-convex rings (yellow) composed of circle segments. Three of the rings contain very acute angles so that hardly any shrinking is possible without self-intersections (1b). It is mandatory to use the intersection-free offset (1c), especially when the sampling rate of the circle segments is increased (1d). The offset operation can be combined with an extrusion (2a), but the elevation can also be downwards when using negative y -values in the profile (2b).

The reversible intersection-free offset permits to expand the polygon to its original size, which also re-creates the three rings that were removed completely by the previous shrinking step (2c). When an expansion to the original size is combined with a negative extrusion, a few applications of killFmakeRH yield the circular window with actual holes shown in (2d). Note that since the vertical profile is symmetric, the front and back sides of the model in (2d) look the same.

Distribution of the edge sharpness flags. The first column (a) with the five images (3a)-(7a) in Fig. 4.79 illustrates the sharpness rules for intersection-free extrusion. Every vertex of the polygon that is to be treated as a corner must carry a temporary ‘corner’-vertex flag. Only vertical (bisector) edges from corner vertices are sharp, all other vertical edges are smooth. The vertex introduced by an edge event is a corner if and only if at least one of the bisectors that meet comes from a corner. This way a crease in the surface is continued, which is a natural behaviour. The sharpness rules for vertex and split events are more complex but similar in that they attempt to continue crease curves. They are consistent so that variations of the sampling rate of smooth boundary curves do not dramatically change the sharpness flag distribution.

A close-up of the first model (a) shows that the three fingers that seem to be stuck into each other near the center are in fact correctly joined by sharp edges (7a). The model in the second column (b) is similar to (a) but it contains a smooth extrusion, with smooth horizontal edges (4b) that are the effect of a *smooth profile point* (x, y, s) with $s = 0$. The resulting subdivision surface is bent in two directions, but it shows only very few wobbles (5b,7b). The horizontal offset is smaller than in the first model (a), so a small hole remains in the three fillets (7b), and unlike the first model, the front- and back-parts of the fillets remain connected. The last column (c) finally combines all these effects and shows what can be achieved with a reversible intersection-free offset, combined with rules for smooth and sharp edges. It uses a much more complex profile, basically the symmetric version of the profile from the classical column basis in Fig. 4.62.

Intersecting tubes by intersection-free extrusion of adjacent faces. When polygons and rings are composed of circle segments, an obvious idea is to use circle segments also for the profile. The first row of Fig. 4.80 shows a profile with a circle segment applied to a polygon similar to the one from Fig. 4.79, only now with six foils and fillets. The nice result in 4.80 (1c) suggests another idea: To join the circle segments on the interior side and the fillet side in order to create a *network of branching tubes*.

But to achieve this the space between center and fillet polygons must be removed as shown in the second row: Instead of inserting several rings into a single front-face, there is only one ring that contains all fields. The seven circles in (2a) are directly connected, and all boundaries that seem to touch indeed share a vertex. The seven circles are then extruded with a symmetric profile, followed by killFmakeRH to turn the extruded faces into rings of the back side face to create the actual holes (2b). When the same is done with all fillets (2c) the result is the desired network of tubes (2d).

With a variety of profiles the impression of a curved front-facing profile remains prominent (3a-d). The close-up of the resulting shapes (4a-d) reveals that despite the fact that touching circles share a vertex, the surface of the tubes is smooth also at the joints. The tubes have no visible artifacts even when the circular profile is densely sampled (5a-d).

4.5.6 Refined Modeling Tools

This section has so far presented an exemplaric collection of mesh modeling tools. They demonstrate the capabilities of combined B-reps and show how the edge sharpness flags can be set automatically, rather than to specify them explicitly edge by edge. The purpose of these tools is also to serve as concrete examples that demonstrate how the clean low-level mesh manipulation interface, the Euler operators, can be used to assemble more user friendly higher-level modeling tools. In particular, the following tools were presented and discussed, now formulated as operations:

- **4.5.1 polygon to doubleface** convert a sequence of points to a double-sided face
- **4.5.2 bridge faces** connect the vertices two face that have the same degree
- **4.5.3 simple extrude** lift a face by a specified amount of elevation, sweeping out side quad faces
- **4.5.4 path extrude** apply a profile to a whole connected path of halfedges
- **4.5.5 intersection-free extrude** apply arbitrary profiles also to non-convex faces

This list is of course by no means exhaustive. It is only a matter of creative fantasy to extend it – and of practical modeling problems that need to be solved. In particular, the following tools may be good candidates for a more complete mesh modeling tool box:

- gluing two faces with the same degree directly together by pairwise merging of their vertices
- gluing together a pair of overlapping coplanar faces with opposite orientation, possibly with rings
- to determine the intersection path of a 2-dimensional plane with a mesh
- to determine the intersection path of two connected components of a mesh, or the set of all such paths
- to cut away one part of an object, as specified by a closed halfedge path
- tools for mesh deformation that include rules for subdividing entities that are too much bent

Some items on this list would also be part of a CSG toolbox, as described by Mäntylä in his book [Män88], to compute the boundary representation of objects created by boolean set operations, i.e., the union, intersection, and difference of 3D solids. But as explained below, it may be preferable to have the individual building blocks of the CSG method available, rather than to completely rely on CSG as the one and only way to model 3D solids.

User-friendly interactive modeling. Of course more than modeling tools is needed for a complete interactive modeler. The first vital ingredient is a flexible mechanism for selection: To specify which objects and which parts of their surface are affected by a modeling operation, and to restrict operations to some selected set of vertices, edges, and faces.

Second, modelers use to support the artist by ‘soft’ techniques that are part of the user interface, such as *snap modes*: While dragging a handle interactively with a 2D mouse the cursor automatically jumps to special locations nearby, such as the intersection of two curves, the foot of a perpendicular, or a tangent point on a curve.

A third very important technique is that interactive modelers offer *intelligent tools* that try to guess what the artist may probably want. This means usually that the tool secretly examines the shape configuration to avoid configurations that might be perceived as inconsistent. One example is the extrusion of neighbouring coplanar faces, shown in Fig. 4.81.

The drawback of intelligent modeling tools is that they are influenced by hidden parameters, which may sometimes lead to an intransparent behaviour. Furthermore they behave differently in different local shape situations, which may be a serious problem, e.g., if a fixed sequence of modeling operations is to be applied subsequently.

Can there ever be enough modeling tools? It is a very interesting question whether any list of high-level modeling tools can in fact ever be exhaustive. The problem is that to some extent, the distinction between a parameterized model and a modeling tool is only arbitrary: The model of a screw for instance can be regarded as a 3D mesh, but also as the result of a specific deformation, namely a *twist* or *lathe* operation.

But note that a screw can also be created by multiple extrusions, each extrusion being followed by a small rotation of the extruded face. So it might well be true that there is only a limited number of generic high-level modeling operations from which most other high-level modeling operations can be composed, depending on the shape domain. If this is indeed the case then the most vital ingredient for efficient 3D modeling will be a facility that permits to flexibly combine existing tools to create new tools from them. One approach to achieve this is presented in the next chapter.

Chapter 5

The Generative Modeling Language GML

This chapter presents the Generative Modeling Language (GML). The GML is a simple stack-based language, and its purpose is to serve as a *smallest common denominator* for the representation of procedural models – similar to the rôle of triangles for representing surfaces. It realizes a paradigm change in low-level shape representations since it uses operations instead of objects: A shape is represented by its generating functions; hence the name *generative modeling*. But the GML is also capable of representing primitive lists efficiently, i.e., it is compatible to legacy data formats.

A central idea of the GML is to combine simple shape construction operations to obtain more involved operations. This solves the problem stated the previous section 4.5.6, the possibly unlimited number of required modeling tools: The GML permits to define *domain-dependent modeling tools*. It also supports and facilitates the maintenance of the tools already produced because it offers the concept of hierarchical tools libraries. It is only by a digital library of modeling tools that existing solutions for specific modeling tasks, and thus the valuable *procedural knowledge* they contain, can be preserved for later re-use.

Originally, the second principal research goal besides better *re-usability* was to improve the *changeability* of intricate shapes and constructions. Usual modelers allow to interactively apply high-level modeling tools until the resulting shape matches the idea of the artist. The GML however is not a modeler but a file format with an interpreter and a runtime component. Unlike other low-level file formats it is capable of representing a description of the construction process. So it permits the artist to express his ideas of how an object is build, rather than only the result of the modeling process.

Every particular shape can be understood as being only an instance of a more general shape class – or of many shape classes in fact, corresponding to the many possible parametrizations. The GML supports the idea of shape classes because it allows to switch back and forth between free and bound parameters. Second, it encourages the separation of input data from processing instructions to obtain the description of a general construction, also from a single given object instance. It can often be even much simpler to generate a GML shape class than a ‘frozen’ instance without any parameters – simply because it is the nature of man-made assemblies that they are built up from similar parts.

5.1 Putting the Pieces together – Why and how realizes the GML the Idea of Generative Modeling?

Perhaps the most important contribution of the GML is that it is a *practical* solution to a theoretically and technically very challenging problem: Although the idea of generative modeling has been recognized for some time as being theoretically very appealing, there is still no suitably successful and broadly accepted concrete realization for it today.

This section elaborates on the question whether, and in which ways, the GML might be a solution to this problem. The argumentation below partly overlaps with what was said before in this thesis. But to better appreciate the rest of this chapter it is important to review what has been achieved so far in a condensed form.

The nature of man-made shape: Structural similarity. Almost all man-made objects contain parts that are similar: Similar in shape, in style, in the degree and resolution of detail, or in function and purpose. A very strong incentive to make things similar is the way objects are physically produced: Industrial manufacturing with assembly lines is rational and efficient only if basically the same means and machines can be employed for the different parts of a product. And then the different parts have to fit together, so standardized styles and measures are also mandatory. Everybody who has ever assembled IKEA furniture knows about this – and about the pros and cons of system furniture.

And structural similarity is even more a prerequisite for today’s *mass customization* development which tries to abandon any static product palette and where every single product item may have a parameter set of its own.



Figure 5.1: Structural similarity that exists in shape data can only be suitably expressed with a procedural model representation. This leads inevitably to a programming language approach for representing 3D shapes.

Primitive-based shape representations fail to capture structural similarity adequately. All shape representations that are based on lists of geometric primitives share the same fundamental problem (see Fig. 5.1). As mentioned in section 1.5 it was most pointedly formulated by Jim Kajiya in the foreword of Snyder’s book ‘Generative Modeling’ [Sny92]:

“With a sculpted surface there’s really no difference between a spoon shape and a chair shape; it’s all a matter of positioning the control points in the right places. But a spoon shape has an inner logic, shared by all spoons – and that logic is completely different from that of a chair.”

The central problem is the absence of semantic information in the model; to mention just a few problems caused by that: Shape features such as sharp edges, planar regions, connected regions and shape segments (segmentation problem), are not explicitly represented but must be retrieved by costly and error prone automatic feature detection methods. A single triangle that is part of a large triangle soup does not know whether it is part of a wall, a door, or a car. Post-processing methods, e.g., automatic simplification and the generation of multiresolution hierarchies, break the model symmetry and destroy the links back to the modeling history. Post-processed models can no longer be edited and changed. The maximum model resolution is limited by the resolution of the exported mesh. To deliver enough resolution also for close detail inspection, curved parts are usually exported in a highly oversampled manner with many millions of vertices.

These problems were formulated for triangle soups but they are shared by all primitive based model representations: points, triangles, spheres, NURBS patches, subdivision surfaces, implicit functions etc. Triangle- or point-based methods are charming because they are broadly applicable, to laser-range scanning data as well as to tessellated CAD data sets. But on the other hand it should be possible to do better with synthetic models, i.e., models created in a 3D modeler or CAD system: It does not seem reasonable to export intelligent objects to a primitive list, and thereby to remove all semantic information that needs to be re-invented whenever the model is to be used for a non-trivial purpose afterwards.

Paradigm shift from objects to operations. The solution to this problem offered by generative modeling is not to store the result of a design process, but to represent a model by the design process itself: Store the tools used rather than the models created. The model can be re-generated from the operations whenever it is needed, and at any resolution that is needed. – This approach has great advantages, but also a number of important consequences, which justifies to call it a paradigm change indeed. Note that it is a true generalization of the primitive based approach, since any static piece of data can also be regarded as an operation, namely a constant one.

The principle of information unfolding. The concern of generative modeling is to help identify *meaningful* degrees of freedom. Most complex constructions rely on a few powerful high-level parameters. Varying the width or height of a building as in Fig. 5.1 has a complex effect on the façade. A triangle mesh has a huge number of degrees of freedom (DOFs), the vertex coordinates and the mesh connectivity. These DOFs are not completely independent, and the vertex positions are not random. Generative modeling shall help to reduce the number of parameters, or even identify the few essential ones, which are then unfolded by a suitable sequence of operations, eventually producing a displayable triangle mesh. This idea can be formulated more generally as the *principle of information unfolding*:

To store only few but powerful data from which a great number of less powerful data can be generated on demand, and as efficiently as possible.

The important rôle of Reparametrization. Whenever a shape is designed, many similar shapes are actually designed with it at the same time. The obvious way to make them accessible is by using parameterized instead of static models. Shapes may be considered ‘close’ only because they use roughly the same underlying design principles, or because they can really be obtained by a slight variation of the parameters.

But a well known problem is that the parametrization of a shape is not unique. A simple shape such as axis-aligned box (AABox) can be specified with 6 floats, via two points (*midpoint, extent*), or via (p_{\min}, p_{\max}) . Suppose only the latter parametrization is available, but it turns out that the center m of the box is actually fixed. Then three free parameters $e = (e_x, e_y, e_z)$ remain for the extent of the box; only they need to feed the box parameters $p_{\min} = m - e$ and $p_{\max} = m + e$. And when the box suddenly needs to be rotated, a 3×4 matrix may be an even better representation.

Consequently the key to reducing the degrees of freedom, as well as to finding the right DOFs, is the possibility to re-parameterize a shape in arbitrary ways. But note that especially for procedural shapes a reparametrization is not always just an algebraic expression, but may have to be procedural as well.

The picking problem as the main obstacle to automization. Three-dimensional shape data are highly structured. Parameterized generative models permit to represent the dependencies between shape parameters explicitly. Furthermore, it becomes possible to concisely represent all kinds of regularity between data items. This is the basis for a gradual transition from a list of primitives to a fully parameterized model. But of course ‘there is no free lunch’: The dependencies must be understood before they can be made explicit.

The problem also with the greatest interactive tools is that the user must use them interactively. It is somewhat ironic that despite the support from very powerful computer technology, to create a three-dimensional object requires a great amount of manual intervention. Why is the degree of automization not higher? One important obstacle seems like a technicality at first, but is in fact fundamental: The picking problem. Interactive selection is extremely hard to automatize since the computer has to guess why which data item (vertex, edge, face, etc.) was picked by the artist. The automatic inference can of course be facilitated when the number of items to pick is (radically) diminished.

Meaningful 3D interaction and the vision of a ‘cyberspace’. The original motivation for the *Virtual Reality Modeling Language* (VRML), later X3D, [VRM97, X3d03a, X3D03b] was to promote the use of 3D technology for all different purposes and application domains; but unfortunately VRML is just *not* a modeling language.

Buzz-words like ‘cyberspace’ and ‘immersive virtual reality’ were popular and hip in the late 1980s. But it turned out that a responsive, entertaining 3D world is quite expensive, and that 3D is all in all a very cost- and labour-intensive technology. It requires skillfull artists to design the world, and much programming to define the behaviour of items and avatars in it. Such an effort is commercially viable only in the computer games sector. But even then, much of the behaviour is static and pre-scripted; all the roads to go and the possible events need to be defined in advance. The interaction with the ‘virtual world’ is in most cases limited to killing virtual enemies.

An unprecedented level of interactivity is possible in VR on the basis of generative technology. When objects are replaced by operations, static geometry can be dynamically re-generated. This permits not only meaningful 3D interaction by manipulating powerful high-level parameters. It opens also an even more thrilling perspective, the interactive assembly of pre-defined shapes and behaviours, and the partial re-creation of a responsive virtual world while being immersed in it.

Abolish the distinction between modeler and viewer. When generative models are displayed interactively it is natural to require that the shape parameters should also be interactively changeable. The important consequence is that the whole modeler must be part of the viewer then, and the separation between authoring tool (model export) and 3D viewer (model import) becomes obsolete. Generative modeling makes sense only with a software architecture that realizes both aspects, a modeler optimized for rendering at interactive rates, and a viewer with full modeling capabilities.

This is technically quite demanding with respect to the underlying shape representation. Snyder, who invented the method, has used composition operators for real functions. But the objective of this thesis is to demonstrate the feasibility and power of generative modeling operating on meshes, which eventually leads to using combined B-reps.

1. **small set of shape generating operators**
 - set should be closed and complete
 - operators should be invertible for undo/redo
2. **expressiveness by meaningful degrees of freedom**
 - powerful: few operations create a complex shape
 - convenient: parameters are intuitive
3. **tessellation leads back to semantics**
 - optimized adaptive display
 - link back to generating function exists
4. **selective updates are possible**
 - amortized preprocessing
 - tessellation on demand

Figure 5.2: Requirements for a shape representation to be suitable for generative modeling. The combined B-reps fulfill these requirements.

5.2	GML generative modeling language
4.4	Progressive Combined B-reps Euler operators and Euler macros
4.3	Combined B-reps B-rep mesh and tessellation data
3.4	Catmull/Clark surfaces adaptive multiresolution rendering

Figure 5.3: Layered software architecture. High-level information is unfolded to create more explicit data from one layer to the next.

Combined B-reps as a paradigmatic low-level shape representation for generative modeling. Generative models are models created by shape generating functions. These shape operators need to operate on some shape domain. So generative modeling requires an underlying shape representation. But not all shape representations harmonize equally well with the generative approach. The desirable properties are summarized in Fig. 5.2.

The **first** requirement is that shapes can be generated with operators at all. This is not always the case; bare indexed face sets for instance (as in Fig. 2.23) only consist of two static arrays for vertices and faces. Arrays of course do offer an operator interface, e.g., to `push_back` and `pop_back` items as with STL vectors. These operators contradict the other requirements, for instance they are not convenient, and they do not support selective updates: An edge collapse requires index updates for two neighbouring faces (linear search without neighbourhood information), and the deletion of a vertex decrements the indices of all following vertices. Consequently insertion and deletion of array elements is feasible only at the end. Despite this limitation this approach is sufficient to ensure the backwards compatibility of generative modeling to legacy shape representations based on lists of primitives (see Figs. 5.61 and 5.60 later). The Euler operators are of course a much better suited mesh interface since they have both properties demanded by the first requirement.

The **second** requirement deals with the principle of information unfolding as another view on generative modeling. The desired DOF reduction can be supported by the shape representation, as exemplarily demonstrated by the combined B-reps. One feature is that they combine polygonal modeling with freeform modeling by the means of a boolean edge sharpness flag. In curved regions a single quad face of the control mesh unfolds to a tessellation with $2 \cdot 16 \cdot 16 = 512$ triangles on the fourth refinement level. But not every freeform shape representation is automatically well suited for DOF reduction. When a NURBS patch is to be deformed at a location where no CV is, an additional DOF must be inserted. But knot insertion leads to a whole new column (or row) of CVs. The possibility of irregular refinement was the reason for choosing subdivision surfaces instead of NURBS for implementing patch complexes (idea from section 2.4).

The **third** requirement is that the tessellation is not a ‘dead end’: It should be possible to find out whether a given triangle belongs to a wall, a door, or a car. This is the technical prerequisite for identifying the generating function of a specific part and also its construction parameters by ray intersection, or by interactive picking with a pointing device. – Note that the requirement to maintain the link back to semantics is technically somewhat demanding when the tessellation is also to be optimized for adaptive display. Adaptivity means that the tessellation must be available in multiple resolutions. But in other terms this means that unlike with automatic simplification a multiresolution tessellation is required that does *not* break the link back to the modeling history. As explained in section 4.4.3 the combined B-reps fulfill this requirement with a multiresolution tessellation that is over and beyond that organized in hardware friendly triangle strips.

The **fourth** requirement rules out shape representations that may have powerful degrees of freedom, but for which the tessellation can not be re-computed at interactive rates, 20 times per second (implicit surfaces, CSG). Fast undo/redo is mandatory for interactive parameter variations. Even if a generative description is available, it is not feasible to completely delete and re-generate a very complex model at interactive rates. This has an important technical consequence: Undo and redo must also be possible out-of-order. So when the description builds independent parts *A*, *B*, and *C* in this order, it must be possible to undo *B*, and to redo *B* with slightly different parameters, without affecting *A* and *C*.

The combination of all four requirements has led to the *layered software architecture* in Fig. 5.3. It is inspired by the principle of information unfolding: The GML issues Euler operators (directly or indirectly) that are stored in pcB-rep Euler macros. Euler macros are the unit for undo/redo, which changes the mesh stored in the cB-rep. And eventually the Catmull/Clark tessellation computes and caches the multiresolution tessellation of the curved surface parts.

Procedural models deserve a procedural representation. Perhaps the most striking consequence of the generative approach is that a data format for an operator-based shape description needs to be a *full programming language*, i.e., Turing complete. Most interestingly 3D modeling and programming have a lot in common. When writing a computer program, a complex task needs to be suitably modularized into well-defined sub-tasks. The resulting function library or class hierarchy should avoid redundancy and provide sufficient functionality to fulfill the task.

Another very concrete argument is that loops and conditional decisions occur very frequently in 3D modeling. They are especially important for highly regular shapes with slight variations. This can be verified, e.g., with the building façade from 5.1, as well as with the examples of linear and circular sequences in Figs. 1.2 and 1.4 from the Introduction.

Furthermore, all interactive shape modeling is procedural modeling. It is an iterative process where an artist interactively applies different modeling tools to manipulate the shape of an object. But the selection of the next operation is not arbitrary; it is based on an idea, on a plan in the artist's mind. This plan usually proceeds coarse-to-fine, and it involves modularization, conditional decisions and branching as well as loops and sub-tasks. But unfortunately the modeling tool does not allow the artist to formulate this plan explicitly. Today's artists are in fact *programming* their shapes, probably without noticing that, and with not much support for it from their modeling software. – The problem remaining is to find a suitable file format standard for procedural models.

There is no exchange standard for procedural models. Most if not all commercial high-end modeling and CAD systems use an intelligent internal model representation, one that encodes also procedural aspects, because of its many advantages (size, changeability, etc.). But unfortunately there is no common exchange standard for procedural models. This makes the exchange of intelligent models between different vendors often impossible. Always possible is exchange only on a low primitive level, for a single 'frozen' instance of an intelligent model.

The reason is not (only) that the vendors deny the usefulness of any common high-level exchange standard. There is also a more fundamental reason. Suppose an intelligent shape a is created in system A and transferred to a target system B , where it is to be used as a component in a larger assembly. The advantage of using an intelligent shape is that it can adapt to changes in the target assembly. But of course a uses the modeling tools of system A , where it was created; and it needs access to these tools when the model parameter change. So to transmit only the construction descriptions is not sufficient; the implementation of the modeling tools needs to be transmitted as well. But this would essentially mean that when a is loaded into B , the system A must be integrated with B , or attached to B . But this would require applications A and B to be compatible on the binary level. To achieve this is definitely illusionary.

This is a serious problem for industrial CAD. One way to alleviate the problem is to define a common minimum standard of modeling operations supported by all CAD tools. This is pursued by the STEP consortium, as it was reported by the leader of the working group XYZ, ZZ, at SMI 2004 in Genova. But the inherent problem of this approach is that CAD vendors seek to gain advantage over their competitors by offering outstanding modeling features.

The code generation problem, and Adobe's PostScript as a solution for it. Most modern modeling software contains an integrated scripting language that allows for the automation of tedious manual modeling tasks. But it is questionable whether a text editor is the ideal device to enter processing instructions for three-dimensional shapes. And of course artists shall not do literal programming. The problem is that most good artists are not also good programmers.

There is one very nasty technical problem when using a programming language as a model representation, namely the question which language paradigm to choose for a file format. This is the *code generation problem*:

- to design a generative shape means to 'write' a computer program
- to save a generative model means to generate syntactically valid program source code
- to load a generative model means to parse and execute program source code, which requires an interpreter

In a first attempt the scripting language *Python* [v*] was used as a model representation; it soon became clear though that generating and manipulating Python code was too cumbersome and error prone, despite the beauty of this language.

The solution was to have a look at approaches that work well for 2D graphics. This turned the attention quickly to the well known *PostScript* document standard from Adobe Inc. Less well known is that PostScript is a programming language. Users are typically not aware that printing a document on a PostScript printer makes the printer in fact to execute a program that, as a side effect, creates the bitmap that eventually appears on a sheet of paper. PostScript as a programming language is quite concisely described in the chapter 3 of the PostScript Language Reference, also known as the *PostScript Redbook* [Ado99]. This fabulous chapter applies, to some extent, also to the GML.

The outstanding feature the GML inherits from PostScript is the simplicity of code generation. This is witnessed by the great number of Postscript printer drivers available. In terms of quantity of automatically generated code, Postscript is most certainly unparalleled by any other programming language.

5.2 Language Introduction

The GML is a very simple *stack based* programming language. The stack, more precisely the *operand stack*, is used for exchanging parameters between functions. Each function pops its input parameters from the stack, processes them, and stores one or more results back on the stack. These results, and the items that were on the stack before, then become the input parameters for the next operation. So technically the job of the GML interpreter is only to shift data from one function to the next, and to feed data produced by one operation into another. Just as bitmaps with PostScript, a three-dimensional shape is generated by the GML only as a side-effect of the program execution.

There are two kinds of functions: Elementary built-in functions implemented in C++, called *atomic operators*, and *composed operators* made of data and calls to other atomic or composed operators. The GML is so simple that it does not even have a grammar. Instead, the language is best described semantically by a dozen simple rules. They are concisely summarized in Fig. 5.5. The tokenizer patterns from rule 2 are also shown in Fig. 5.4.

5.2.1 The Language Rules

The first very basic example shall only illustrate how the GML rules work together. It reads “2 3 add 4 mul \rightarrow 20”.

This notation means that the code to the left, when evaluated, produces the value to the right on the stack. As far as the stack is concerned, both pieces of code are equivalent.

A stack is usually thought of as growing upwards, like a pile or stack of real items, where every item naturally lands on top of the pile. While still retaining this metaphor, a horizontal rather than vertical notation is often more useful, where the stack grows *to the right* and the ‘stack top’ is the rightmost element. In this case the stack can be directly read as program code. Both pieces of code are readily understandable as being ‘stack equivalent’, i.e., they produce the same result on the stack. But of course they do not need to have the same side effects, e.g., they may not produce the same 3D shapes.

Execution according to the rules. When the code of the GML program 2 3 add 4 mul is tokenized, the result is an executable array of five tokens (rules 1,2). To execute this program then means to execute each of them one after another (rule 6a). First the tokens 2 and 3 are executed. They are literals (rule 2), so they are just pushed (rule 6b). The token add is an executable name (rule 2) that, when executed, is looked up (rule 6c). One dictionary is by default on the dictionary stack (rule 7), the *global dictionary*. Under the name add it stores the *add-operator*, which is then executed for the name (rule 6c). The *add-operator* is atomic, so it pops its two arguments and pushes the result $2 + 3 = 5$ (rule 12). Then the literal 4 is pushed, the *mul operator* first pops the stack top 4 and then the 5, and finally pushes the end result $4 \cdot 5 = 20$.

The advantage of the postfix notation is that no brackets are needed for arithmetic expressions. The familiar infix notation $(2 + 3) \cdot 4$ needs brackets because of the precedence of \cdot over $+$. All operators have equal rights in postfix notation, also called *reversed polish notation* (RPN) [wika], which has therefore been quite successfully used in commercial and in pocket calculators, such as the famous HP48 from Hewlett-Packard [R*].

Almost any string can be tokenized. One consequence of the tokenizer rules 1 and 2 from Fig. 5.5 is that the tokenizer accepts almost any character string as GML code. The reason is the fallback rule that any coherent sequence of non-stop characters is just tokenized as an executable name. This means that variables or functions may be referred to with very uncommon identifiers, as demonstrated in Fig. 5.4. Only the two extensions to the original PostScript syntax, path names and registers, make the tokenizer a bit more rejective. Still there are only five syntactic errors:

- unterminated string error: an odd number of " characters
- unterminated function error: more } than { characters
- prefix error: prefix character (. / ! : ;) followed by whitespace
- register usage error: no usereg appears before the first !name
- register not set: :name appears before !name

22	\rightarrow	int	(1.1,2.2)	\rightarrow	2D point
23.2	\rightarrow	float	(1.1,2.2,3.3)	\rightarrow	3D point
"hallo"	\rightarrow	string	/name	\rightarrow	literal name
[{ }	\rightarrow	marker	.name	\rightarrow	path name
<hr/>					
+n#a#m-e*	\rightarrow	executable name, fallback rule if everything else fails			

Figure 5.4: Tokenizer patterns for literal tokens.

1. GML code consists of individual tokens, separated by stop characters such as whitespaces.

The GML has no parser but only a tokenizer to convert ASCII text into an executable array of fixed-size tokens. Further stop characters are one-character tokens [,], { , }, the / and . start literal and path names, " encloses strings, and ! , ; , : are for registers. A comment starts by % and is read, with the rest of the line, as a single newline.

2. Every token has a unique type from a fixed but extensible type set.

Built-in atomic data types are *integer* 123, *float* 123.45, *vec2f* (1.2,3.4), *vec3f* (1.2,3.4,5.6), *marker* [, { , }, *string* "abc", and *literal name* /myname. The type set can be extended on the C++ level where new tokenizer rules can be added. The other builtin token types are *dictionary*, *array*, *operator*, *path name*, and *name*. A token for which no other rule applies gets the type *executable name*, and is entered into the global (*string*, *ID*) map.

3. The only built-in compound data structures are arrays, dictionaries ('dicts'), and strings.

An array is a heterogeneous token sequence, i.e., the individual tokens may be of any type. Arrays are cyclic, so index -1 refers to the last, and index n to the first element of an n -element array. The first element has index 0. A dictionary is a map with (name,token) pairs where the name (i.e., its *ID*) is a unique key.

4. Tokens for compound data types contain only a reference.

All tokens have the same size, 16 bytes = 128 bit = 4×32 bit. Therefore tokens for variable size data types contain only a reference to the actual data. This applies to the built-in types array, dictionary, and string, as well as to names: The GML interpreter maintains a global (*string*, *ID*) map that is used to tokenize all names.

5. Tokens can be either *literal* or *executable*.

Atomic and user-defined tokens and also dictionaries are always literal. Operators and path names are always executable. Name and array tokens may be both literal or executable, depending on their *executable flag*.

6. (a) To execute an executable array means to execute each of its tokens, from begin to end.

(b) To execute a literal token simply means to put it on the stack.

(c) To execute an executable name means to look it up and to execute the object found.

7. Flexible scoping: Name lookup uses the *dictionary stack*.

The topmost dictionary on the dictionary stack that contains the requested name as a key also defines the value. Dictionaries can be pushed on and popped from the dictionary stack at any time with the operators begin and end.

8. The opening marker [and the closing bracket operator] create literal arrays.

When the] operator is executed it looks through the operand stack for the first [marker. It pops all tokens in between, puts them into a literal array, pops the marker, and pushes the array (i.e., an array token that refers to it).

9. From each matching pair of { and } brackets an executable array is created.

An executable array is also called a *function*. The first { marker puts the interpreter in *deferred mode* so that it treats *all* tokens as literals and puts them on the stack, until it finds the matching } (pairs can be nested). The resulting executable array is only pushed on the stack, not executed.

10. Path name extension: Navigation in dictionary hierarchies.

The semantics of the dot prefix is that when a path name .myname is executed, it pops a dictionary from the stack, and the token that is found in it under myname is pushed. So it replaces the dictionary as topmost stack element, which makes C++-like path expressions man.arm.hand.finger possible.

11. Named register extension: !x pops and stores, :x executes, and ;x pushes the value of register x.

Registers can only be used *within* a function, after a new register frame was opened with the usereg operator. The register frame ends automatically with the function. Registers are faster than dictionaries or the operand stack.

12. All functionality comes from atomic operators.

An operator is executed by calling the execute method of the appropriate C++ class. It pops its inputs from the stack, processes them, and pushes the results back on the stack. The GML interpreter has no fixed set of keywords or any built-in functions. It only manages an extensible set of operators, organized in libraries (dictionaries).

Figure 5.5: The twelve GML rules. A dozen rules are completely sufficient to describe the language.

Tokens	cvlit cvx exec type
Stack	clear cleartomark count counttomark dup exch index pop roll
Arrays	aload append array arrayappend length popback
Dictionaries	def undef dict begin end currentdict keys known values where load
Arrays and dictionaries	copy get put
Comparison	eq ne ge gt le lt
Flow control	if ifelse repeat for forall map loop exit

Figure 5.6: Essential GML operators (core library). They behave just like the respective PostScript operators.

GML language core. The GML interpreter has no built-in functions; all functionality comes from operator libraries. The core library contains the basic atomic operators to make the language work, shown in Fig. 5.6. Not each of them is discussed in this section, for a detailed specification refer to the PostScript Reference [Ado99].

Since the principle of a stack-based language is at least mentioned in most introductory courses on programming, the basics should be known: `count` pushes the stack depth, `clear` removes all items from the stack, `cleartomark` and `counttomark` do the same until the first `[-`marker, `dup` duplicates the stack top, `exch` swaps the top elements, `k index` pushes a copy of the k 'th stack element. `pop` removes the stack top, and `roll` is essential for PostScript but practically never used in the GML. Important array operators are `[1 2 3] aload` \rightarrow `1 2 3` to put array items on the stack, `[1 2] dup 3 append` \rightarrow `[1 2 3]` to append an element, `[1 2 3 4] 2 popback` \rightarrow `[1 2]` to remove the last 2 array elements, `arrayappend` to concatenate arrays, `length` pushes the array length, `[10 11 12] 1 get` \rightarrow `11` retrieves, and `put` replaces an element.

Arrays and inline computation. GML arrays are heterogeneous, so an array may contain tokens of any type. An example of a valid array is `[22 4.7 (0,1) (2.2,3.3,4.4) "hi"]`. There is an important difference between the starting and ending brackets: `[` is a literal marker but `]` is an operator. The `[` is a usual token, and it can be pushed, popped, duplicated, exchanged, and stored in variables just like any other token. In particular it is legal to push any number of `[` on the stack. An array however is created only when the array construction operator `]` is executed. At this point, at least one `[` must be on the stack, so that an array can be created from the items in between; otherwise the program execution stops with an unmatched mark error. The `[` may the stack top element, so `[dup]` creates an array that contains another empty array.

The array creation operator pushes an array token on the stack. As explained in rule 5, there are no variable-sized tokens; all tokens have the same size of 16 bytes. A token can only *refer* to a variable size data structure, so array and dictionary tokens behave more like *pointers*; they contain just a type flag and the ID of the respective array or dictionary. The manipulation of arrays and dictionaries is therefore always *by reference*. There is no overhead when pushing arrays, dicts, or strings on the stack.

- In `[1 2 3] dup` \rightarrow `[1 2 3] [1 2 3]` the duplication has exactly the same cost as in `12 dup` \rightarrow `12 12`
- Changes have an impact on all references to the same array: `[1 2 3] dup dup 1 20 put` \rightarrow `[1 20 3] [1 20 3]`

Naturally arrays can be nested, so `[1 2 [3 4] 5 6]` is just an array of length five. Its tokens 1, 2, 3 (and also 6, 7, 8) have the types integer, array, integer. Note that GML arrays start with index 0, and they are cyclic (rule 3). One of the extremely convenient features the GML inherits from PostScript is *array inline computation*: `[4 5 mul 6 7]` \rightarrow `[20 6 7]`

Executable arrays are functions, and functions are executable arrays. Just like PostScript the GML has a remarkable property: It is a functional language, in the sense that functions are ‘first level citizens’. Functions are just one data type among others; functions can be both parameters and return values of other functions, new functions can be created at runtime, and existing functions can be changed. Technically this is achieved by providing arrays (like names) with an *executable flag*. This makes it possible to switch between data and functions in a very simple way, since both are implemented with arrays. So a freshly assembled array of data can be turned right away into a function using `cvx` (‘convert to executable’) and back with `cvlit`, which both just affect the executable flag.

```
[ 1 2 3 ] cvx   →  { 1 2 3 }           { 1 2 3 } exec →  1 2 3
{ 1 2 3 } cvlit →  [ 1 2 3 ]         [ 1 2 3 ] exec →  [ 1 2 3 ]
```

Note that there is a subtle difference when being executed: A literal array is just pushed on the stack, whereas an executable array is executed token by token. Note that in case all array tokens are literals, to convert an array to a function and to execute it is the same as to apply `aload` to the array: All items are individually put on the stack. – The great advantage now is that all array manipulation operations can be equally applied to functions:

```
{ do some thing } { and something else } arrayappend → { do some thing and something else }
{ do some thing 14 times } dup 3 1000 put             → { do some thing 1000 times }
```

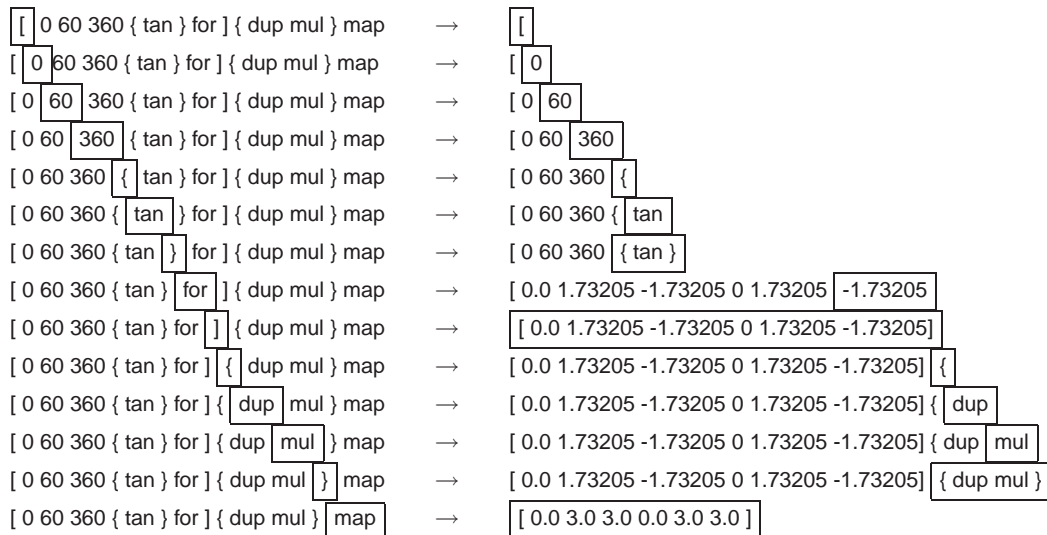


Figure 5.7: Step by step execution of the GML code [0 60 360 { tan } for] { dup mul } map .

So the output of a GML program can be a processing pipeline that is defined and assembled at runtime. The construction history of a shape can for instance be obtained by concatenating the individual processing steps; the result is a single function. Even self-modifying code is possible (but maybe not so useful).

Flow control and function calls. Loops and conditional branching are essential features for any programming language. Unlike other languages such as C/C++ the GML does not need any special syntax rules or reserved keywords ‘if’ and ‘for’. Instead, flow control can be elegantly realized with *flow control operators*, thanks to the flexible array mechanism. The branching operator is an ordinary atomic operator with the signature ‘*flag proc if* → -’. The *signature* is a semi-formal notation that describes the effect of an operator to the the stack. It means that the if-operator takes two tokens from the stack, first a ‘procedure’, then a numeric flag. As one might expect *proc* is executed by the if-operator only in case the flag is not zero (integer or float).

The GML feature used here is that operators can execute tokens, including executable arrays they pop from the stack. This is just the same as a function call (see below). But note that the *proc* token does not have to be a function at all; it can as well be any literal. The same feature was used above with the *exec*-operator; the only difference between *if* and *exec* is that the latter executes the token unconditionally. The other flow control operators follow the same approach.

<i>flag proc if</i>	→ -	executes <i>proc</i> if <i>flag</i> ≠ 0
<i>flag proc1 proc0 ifelse</i>	→ -	executes <i>proc1</i> if <i>flag</i> ≠ 0 and <i>proc0</i> otherwise
<i>proc loop</i>	→ -	continues to execute <i>proc</i> until the <i>exit</i> -operator is called
<i>n proc repeat</i>	→ -	executes <i>proc</i> <i>n</i> times
<i>a s b proc for</i>	→ -	Pushes values <i>a</i> , <i>a + s</i> , <i>a + 2s</i> , ... on the stack and executes <i>proc</i> each time, until the value > <i>b</i> (if <i>s</i> > 0) or value < <i>b</i> (in case <i>s</i> < 0)
<i>[a1 .. an] proc forall</i>	→ -	pushes each element on the stack and executes <i>proc</i> each time
<i>[a1 .. an] proc map</i>	→ [b1 .. bn]	pushes <i>a1</i> , executes <i>proc</i> , pops <i>b1</i> , then pushes <i>a2</i> , executes <i>proc</i> , pops <i>b2</i> , and so forth; this maps one array to another array

One example of a for-loop in the GML is `0 1 3 { 2 mul } for` → `0 2 4 6`. Note that none of the flow control operators except *map* push return values (‘→ -’ notation), but of course *proc* may leave values on the stack: `0 0.5 2 { }` for → `0.0 0.5 1.0 1.5 2.0`. The decision flag for *if* and *ifelse* is usually generated by comparison operations like *gt*, *lt*, *ne* which read ‘greater equal’, ‘less than’, ‘not equal’ (see Fig. 5.6). Their result is an integer as the GML has no Boolean type.

The step-by-step execution of a slightly more complex piece of GML code using *for* and *map* is shown in Fig. 5.7. As a detail note that the construction of executable arrays via the deferred mode (rule 9) is a large overhead. It means that the nested if-clause in `20 { dup 10 gt { dup mul } if } repeat` is created 20 times. In the actual implementation the function is created only *once*, by the tokenizer and not at runtime: When the tokenizer reads a { it calls itself recursively. A reference to the executable array it produces at the matching } is directly inserted into the calling function. – The execution rules are slightly changed to behave exactly as described in Fig. 5.5, just more efficiently.

Flexible scoping and name lookup. In C/C++ each function call opens automatically a new scope for locally defined ‘automatic variables’. This is not so in the GML, where a scope can be started and ended only explicitly, completely decoupled from any function calls. It may even be that one function begins and another ends a scope.

- the `dict` operator pushes a new empty dictionary
- the `begin` operator pops a dictionary from the operand stack and pushes it on the dictionary stack
- the `end` operator pops the current dictionary

The *current dictionary* is the topmost dictionary on the dictionary stack. It is queried first when a name is looked up. It can be modified with the `def` operator which pops a literal `/name` and a value token to make `(name,token)` an entry in the current dictionary for subsequent name lookup. Note that when a name lookup leads to an executable array, it is immediately executed (rules 6c+6a). This is a *function call* in the GML, which can also be nested.

The following examples show varying the scope with dictionaries. The second example demonstrates temporarily *overwriting* the name `x`; the third how an ‘often used’ operator sequence, `dup mul` which multiplies the stack top with itself, is turned into a function. Note that there is no difference between using a built-in or a user-defined function. Examples four and five are equivalent, both define a function `abs` to compute a number’s absolute value. The load operator pops a (literal) name, looks it up, and *pushes* the value. This is different from an executable name lookup which *executes* the value. This makes a difference in case the value is executable, such as the `mul` operator.

```
dict begin /x 20 def x x end           → 20 20
dict begin /x 20 def dict begin x /x "hello" def x end x end   → 20 "hello" 20
/sqr { dup mul } def 4 sqr 5 sqr sqr  → 16 625
/abs { dup 0 lt      { -1 mul }      if } def 2 abs -13 abs    → 2 13
/abs { dup 0 lt     -1 /mul load 2 array cvx  if } def 2 abs -13 abs → 2 13
```

Names are used as identifiers and can of course have any length. The GML interpreter maintains a global map with a unique integer for each character string that has appeared (so far) as a name. The tokenizer uses this map to convert the variable-size name string to a fixed-size name token. This assures that no string operations occur for names during program execution; furthermore dictionaries can then be efficiently implemented as `map<int,Token>`.

PostScript permits to redefine (overwrite) built-in operators. This is not easily possible with the GML because the tokenizer performs what is called *early binding* in PostScript (section 3.12 in [Ado99]): Every executable name of an operator is immediately replaced by (a reference to) the operator itself. The reason is that this gives substantial speed-ups, and speed is an important issue for the GML because it is used interactively.

The named register extension. Some people think that stack-based languages are horrible to program because *stack acrobatics* was inevitable. A serious objection to PostScript as a programming language is that it can be a nightmare to keep track of the stack, which items it contains and in which order. PostScript offers basically two methods to store a value for later use: On the stack, which can be tedious to keep track of, and in dictionaries, which is relatively slow.

Named registers were added in the GML as a third alternative. They are very efficient, register `set/get` is faster than any stack rolling. Other than dictionaries, registers are bound to the call stack (5.5, rule 11), like the automatic scope in C++. Their main purpose is to provide functions with fast local variables, and they improve both the readability and the efficiency of GML functions. It is good programming style to begin a function by popping all needed parameters into registers. This convention makes also the (**reversed**) signature of the function explicit. Values needed later, intermediate values, and values needed multiple times in a function are all stored in registers. This leads to a much more procedural programming style than in PostScript, and the infamous roll stack operator is practically never used in GML.

The most important thing about registers is that they are valid only *within* a function. Whenever a function ends the interpreter looks whether a register frame was opened in it with `usereg`; this frame is then discarded, and all register values are cleared. The general rule is that *no reference to a named register may leave a function*. Only weird examples such as the following violate this rule: `{ 12 usereg lx { :x } } exec exec` The result is not 12 but an error.

Object-oriented programming (OOP) and the path name extension. Just like arrays, dictionaries also serve multiple purposes. Although it may sound weird at first, they are the basis for realizing an object-oriented programming style in the GML. In an object-oriented programming (OOP) language like C++, a class is a mixture of data and operations. Through inheritance, a derived class can add data and functions, and overwrite virtual member functions. Exactly the same is possible with GML dictionaries. Dictionaries are in fact even more flexible than C++ classes since both data members and member functions may be added, deleted, and modified at runtime. Class members, i.e., dictionary entries, may even switch between being just literal data or functions, as shown above.

GML path names permit to navigate through complex dictionary hierarchies in a familiar way. A tool, e.g., from a library for modeling Gothic architecture can be used with `Gothic.Window.Tools.pointed-arch` (see 5.4.1). With relative names, the ‘modeling vocabulary’ can even be switched: `Arabic.Window.Tools begin ... pointed-arch ... end`

5.3 Examples for Shape Generation with Mesh Modeling Operators

One of the main incentives for a stack-based modeling language comes from the code generation problem and the resulting dilemma: On the one hand procedural models need to be represented by a programming language, on the other hand to program a shape in a text editor is not an ideal user interface.

But shape modeling and programming have a lot in common. This section seeks to point out what these similarities are, and it works out a number of concrete examples. This uses the programming approach indeed, but more as a formal calculus than as a user interface. It is important to examine the expressiveness of the GML to assess its suitability for representing descriptions for shape construction. Is it really suitable as ‘smallest common denominator’ for procedural models?

This is a very general question, and the answer to it depends also on the shape representation. This chapter focuses on the question whether the GML is suitable for mesh modeling. This uses the combined B-reps as low-level shape representation. A number of operators from the CReP library of the GML are therefore introduced in this chapter, as well as the literal types Halfedge and Eulermacro added by this library.

5.3.1 Example Operator Chaining and Creating a Loop

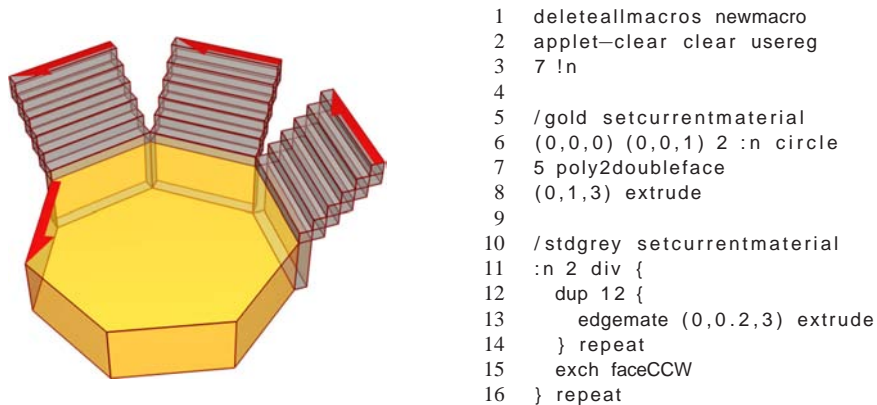


Figure 5.8: Basic modeling techniques ‘operator chaining’ and ‘loop creation’.

Halfedges and basic modeling tools from the CReP library. This first example presents two basic techniques, a simple way to create a mesh and the iterative development of a loop. – It was already mentioned that the GML type set is extensible. All mesh operations use the custom type Halfedge from the CReP library of the GML. Its string representation is, e.g., E19,2,23423, which identifies uniquely one halfedge of a mesh. From the point of view of modeling though, the three integers are basically random as they are different every time a function is called (see section 4.4.3 on stable edge references: index in euler sequence, macro index, timestamp). So halfedges may be stored, compared, and retrieved, and they may appear on the stack. But they may not appear in the GML code.

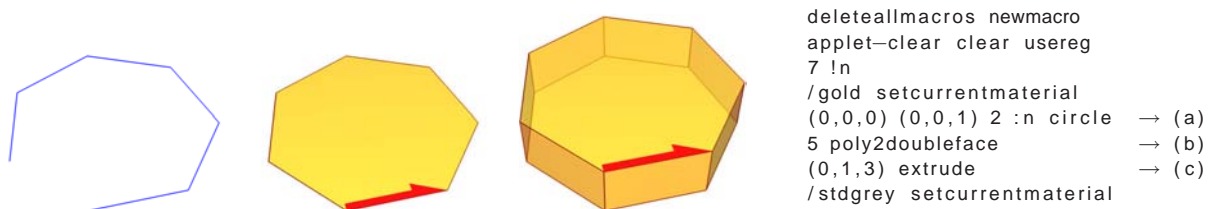


Figure 5.9: A simple way to create a solid.

Standard startup lines. The first two lines of code are the standard way to clear all visible items in the runtime engine: The mesh is reset by deleting all Euler macros and a new Euler macro is started that logs all modeling operations to come. All applets are cleared; applets are special GML resources that can be plugged into the GML runtime engine as extensions. The stack is cleared since by default the runtime engine displays all stack items that are renderable: points, polygons, halfedges etc. Finally usereg starts a new register frame, which concludes the two startup lines.

Operator chaining. The three lines 6, 7, 8 from Fig. 5.8 demonstrate an essential benefit of the stack-based approach: Data can be processed like on an assembly line, the output of one operator is fed into the next. Since the stack is such a flexible device for parameter passing, the GML assembly line works without any explicit declaration of routes from output-results to input-parameter.

A simple way to generate a mesh is to create a polygon (a), to convert it to a double sided face (b), and to extrude it (c). A polygon in the GML is just an array of 3D points, for instance created by the circle operator. The polygon produced by it is automatically given to poly2doubleface which turns the point array into a halfedge. The extrude operator replaces this halfedge by a halfedge of the extruded face.

`center nrml rad arcn circle` → [p1 .. pn] Creates a circular n -gon around center with radius rad in the plane normal to nrml. rad may be a float. It may also be a vector, then its projection to the plane is the first point of the polygon. n equals arcn when it is an integer; when it is a float it specifies the arclength and thereby n .

Mode flags and compatible signatures. When designing a new operator or function the prescribed order of the input parameters be freely chosen. One signature is better than another if it is easier to memorize and more convenient. Many operators have one *principal parameter* and a number of *mode flags*. The principal parameter is the one that was most probably the result of another operator. It should come first in the signature, followed by the mode flags. Operator chaining is most efficient with a set of operators that have *compatible signatures*.

The mode flag of the poly2doubleface operator determines the edge sharpness and the temporary vertex flags of the new face (see section 4.5.1). It is only an integer, whereas the polygon is more complex and has in many cases been produced by another operator. Similarly, mesh operators both operate on and produce halfedges; therefore their principal parameter is the halfedge. An indication for incompatible signatures is that they require often inconvenient stack reordering.

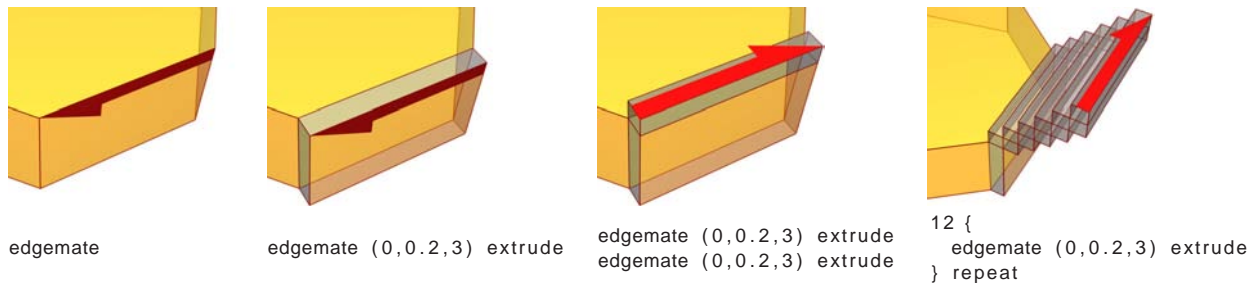


Figure 5.10: Designing a loop step by step. The loop body is created first.

‘Iterative’ loop design. A procedural representation is most useful for shapes that exhibit sequential regularity. Its obvious translation into the generative formalism is a loop. In the GML loops should be created *body first*: When finished, the body of a loop should leave the stack in a state similar to the state before. But what means ‘similar’ in each case?

Halfedges are quite useful for mesh traversal using the navigation operators edgемate, vertexCW, vertexCCW, faceCW, and faceCCW from section 4.1.3 (for B-reps see Fig. 4.19), which all expect a halfedge on the stack and replace it by another. Fig. 5.10 above shows an edgемate (a) followed by an extrusion (b). When this is done twice (c) it leads to a *similar* situation as before in Fig. 5.9 (c). So this line is a good candidate for a loop body 5.10 (d).

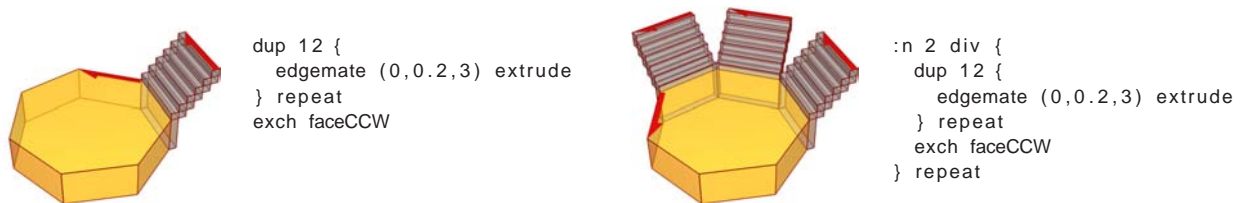


Figure 5.11: A stack item is put aside using dup. After a first round of processing it is made current again.

Branching assembly lines. The stack permits of course not only linear processing. At any point a stack item can be duplicated, which can mean to put it aside for later processing. The loop developed above, even when it is repeated 12 times, just transforms the halfedge on the bottom to the halfedge on top of the staircase. So if the bottom halfedge is put aside before the loop, it can be simply got back using exch after it. A single faceCCW creates a *similar* situation as before, and is consequently a good candidate for the body of another loop.

5.3.2 Example Segment Intersection

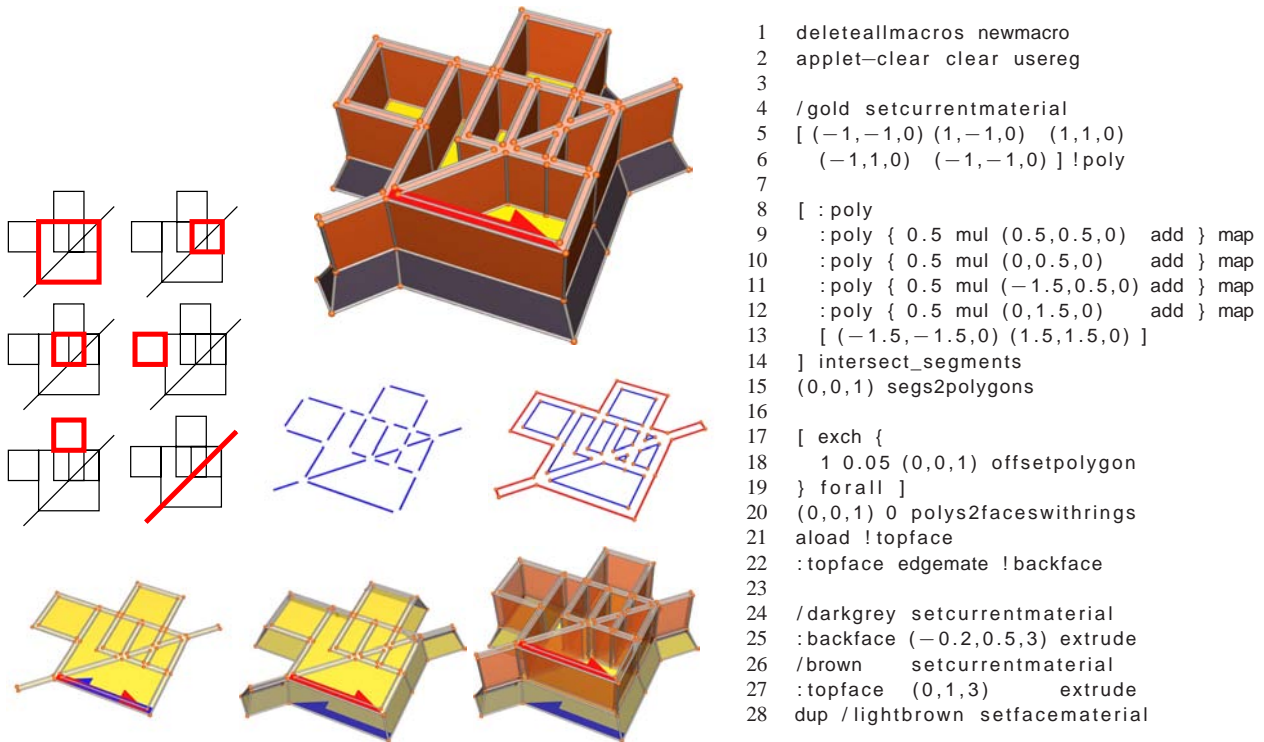


Figure 5.12: Segment intersection followed by offset polygon.

Variations of polygon construction. The statement that ‘polygons in the GML are arrays of 3D points’ is a practical rather than formal rule. It is just convenient to provide operators performing an algorithm on simple polygons with an interface that expects one – or more – point arrays as input. This permits to take advantage of the ‘syntactic sugar’ for arrays, inline computation and mappings. Both array features are used in the example shown in Fig. 5.12. After the standard startup and the material setup, in lines 5-6 a closed quad polygon is constructed and stored in the named register `:poly`. The inline computation in lines 8-14 results in an array containing the six simple polygons shown to the left, in 5.12 (2a): `:poly`, four displaced copies of it, and a line. The `map` operator is applied to all points in `poly`, and the body performs an affine transformation, a scaling by 0.5 followed by a translation.

Operators can be polymorphic. One and the same operator may behave differently depending on the type, and also of the value, of the parameters it finds on the stack; like in OOP such GML operators are called *polymorphic*.

$n0\ n1\ \mathbf{mul} \rightarrow n2$ product $n2 = n0 \cdot n1$ of integer or float numbers $n0, n1$
 $n\ v\ \mathbf{mul} \rightarrow w$ product $w = (n \cdot v_x, n \cdot v_y, n \cdot v_z)$ of scalar n and vector v
 $v\ n\ \mathbf{mul} \rightarrow w$ product $w = (n \cdot v_x, n \cdot v_y, n \cdot v_z)$ of scalar n and vector v
 $v\ w\ \mathbf{mul} \rightarrow n$ dot product $n = v_x \cdot w_x + v_y \cdot w_y + v_z \cdot w_z$ of two vectors v, w

Note that even the number of parameters an operator pops may vary as a function of types and values of the stack items; so operators can also have a *dynamic signature*.

A line segment intersection algorithm. Note that the `:poly` array contains five points and not just four, the first point is repeated. This is redundant for polygons since GML arrays are cyclic. The reason is a different use of point arrays:

`[[P0][P1]..[Pn]] intersect_segments → [p0 q0 p1 q1 .. pm qm]`

computes line segments with startpoints p_i and endpoints q_i from input segments $P_0 \dots P_n$. The output segments mutually overlap at most at their end points, and their set-theoretic union equals the union of the input segments. The input segments may contain more than two points, i.e., may they specify a line strip.

So the line segment end points are all contained in a single big array, they are shown (shrunk) in Fig. 5.12 (2b).

<code>p0 p1 makeVEFS</code>	\rightarrow e	new shell	<code>e killVEFS</code>	\rightarrow -	delete shell
<code>e0 e1 p makeEV</code>	\rightarrow e	vertex split	<code>e killEV</code>	\rightarrow -	edge collapse
<code>e0 p makeEVone</code>	\rightarrow e	dangling edge	<code>e0 makeEFone</code>	\rightarrow e	dangling loop
<code>e0 e1 makeEF</code>	\rightarrow e	split faces	<code>e killEF</code>	\rightarrow -	join faces
<code>e0r e1f makeEkillR</code>	\rightarrow e	attach ring	<code>e killEmakeR</code>	\rightarrow e0r	detach ring
<code>r makeFkillRH</code>	\rightarrow -	ring to face	<code>e0 e1f killFmakeRH</code>	\rightarrow -	face to ring

Figure 5.13: Euler operators as GML operators. The identifiers e0, e1, etc. are identical to those used in the Euler operator diagram Fig. 4.47.

Line segments to polygons, and sorting polygons to obtain faces. The result of `intersect_segments` is another example of a point array that is not a polygon. It can be converted to polygons though by the `segs2polygons` operator. This operator turns a set of line segments into a set of polygons: The line segments partition the plane into disjoint regions, and the boundaries of these regions are polygons. They are computed using a sweep-line algorithm. Besides the line segments `segs2polygons` expects the normal vector of the plane into which the segments are projected (line 15) and in which the regions reside.

Lines 17-19 introduce another technique to create an array `1 [exch] \rightarrow [1]`, which can also be used in a variant, such as `[1 2] [exch aload 3] \rightarrow [1 2 3]`. In lines 17-19 it could actually be replaced by a `map`, though.

The ten polygons produced by `segs2polygons` at first share the same segments. They become disjoint when they are offset to their interior by the `offsetpolygon` operator, as shown in Fig. 5.12 (2c). It produces an array of polygons that then needs to be turned into a mesh. This is a complex task since it involves to find out which polygons are contained in which other polygons. In the example the result is the face with nine rings shown in 5.12 (3a). The face is stored in the register `:topface`, its backside mate (blue halfedge) in `:backface`. The `:backface` has no rings, which can also be seen from the extrusion in 5.12 (3b).

Complex algorithms with simple interfaces. This example has shown alternatives to the usual pipeline for creating a solid, `polygon \rightarrow double-sided face \rightarrow extrusion`. Variations exist especially for creating polygons. The current example demonstrates: `line segment intersection \rightarrow region boundary polygons \rightarrow offsetpolygon \rightarrow sorting into topfaces/rings`.

Also non-trivial algorithms can have quite simple interfaces. The key to a greater flexibility in setting up processing pipelines is a careful design of compatible signatures of a variety of algorithms. In particular the following two operators can be quite flexibly used and are therefore more often employed.

`[p0 .. pn] closed offset nrml offsetpolygon \rightarrow [q0 .. qm]`
 returns the polygon with edges offset by distance `offset` to the left with respect to the plane normal `nrml`. If `closed` is 0 then the point array is treated as line strip, otherwise as closed polygon.

`[[P0] [P1] .. [Pn]] nrml backrings polys2faceswithrings \rightarrow [e0 .. em]`
 turns the simple, non-intersecting input polygons into double-sided faces and sorts them according to the containment relation: If face `r` is directly contained in another face `f`, then the backside of `r` is made a ring of `f`. Iff `backrings` is not 0, then rings are inserted also in the backsides, and rings of rings become again topfaces, and so forth.

5.3.3 Example Cube with a Hat using Raw Euler Operators

This example presents the solution of a problem that was stated in chapter 2. There Fig. 2.23 shows a ‘cube with a hat’ represented as indexed face set using the VRML and .obj file formats for comparison. A fundamental problem of indexed face sets was mentioned in section 2.2, namely the missing semantic information. This problem was the motivation for the introduction of Euler operators in this section.

The generative description of a shape? It is important to note that the representation of a shape as an indexed face is unique – except for entity permutations and surface approximation quality, of course. So formally, a particular shape corresponds to an equivalence class of indexed face sets.

For generative models it is not so easy to determine whether two given shape descriptions are equivalent. The generative realization of the cube with a hat from Fig. 2.23 presented in this section is only one among several possibilities. One and the same shape can be reasonably partitioned into functional or aesthetic components in many different ways, resulting in completely different generative descriptions. So there is no such thing as *the* generative model of a shape. Still some generative representations are more useful than others: Those that are efficient and can be re-used.

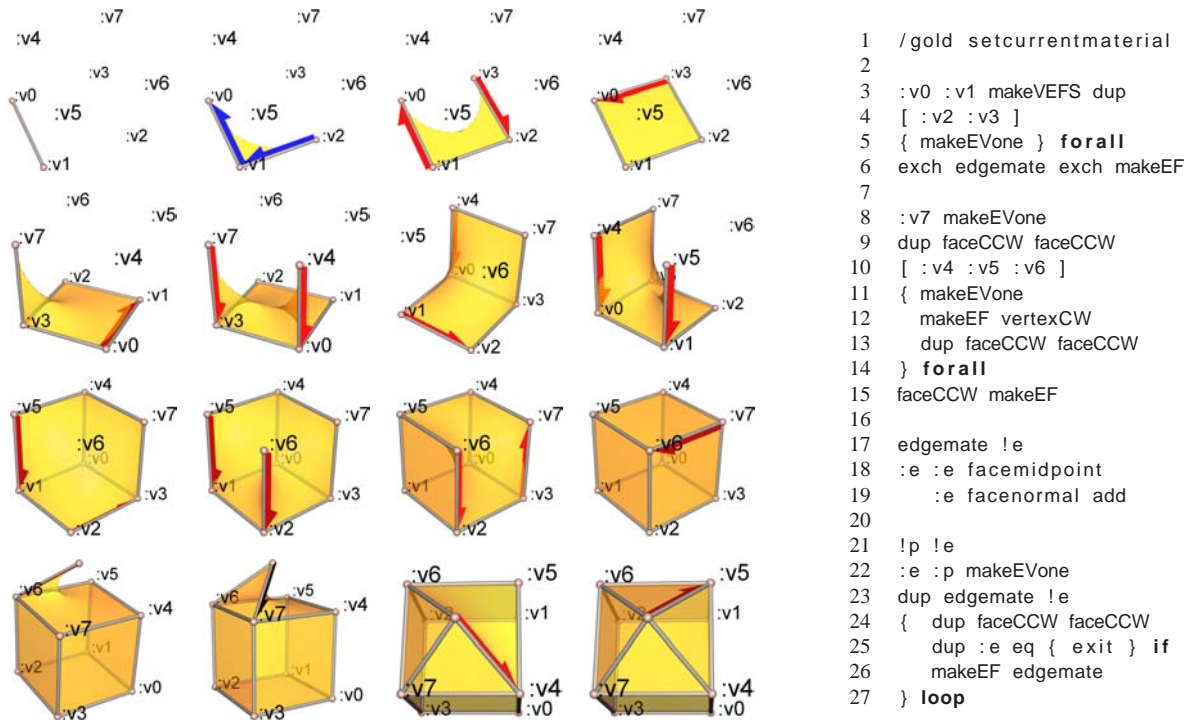


Figure 5.14: Building a cube with Euler operators. Its topology equals the VRML/.obj example from Fig. 2.23, but unlike the VRML example it uses loops which makes it generalizable. The sixteen images correspond to the result of the following source lines: (3,5,5,6), (9,11,13,11), (13,11,13,15), (22,26,26,26).

Modeling with Euler operators. This example demonstrates modeling with Euler operators directly, not as before hiding them in higher-level tools such as `poly2doubleface` or `extrude`. The GML versions of the Euler operators are tabulated in Fig. 5.13. There are five Euler operators and five inverse operators, i.e., a total of ten. For convenience `makeEVone` and `makeEFone` are included to create dangling entities (Fig. 4.47, (1b), (3b)). An Euler operator can create at most one edge, consequently the lines 3-14 issue in total twelve of them, shown in the first twelve images in Fig. 5.14.

Two loops to build a cube. The GML code for the cube with a hat is shown in Fig. 5.14 (right). The creation of the cube part is partitioned into two phases, both of them are loops: Creating a double sided face in lines 3-6 (images (1a-d)), and the extrusion in normal direction in lines 8-15 (images (2a-d), (3a-d)).

The double-sided face is created by a sequence `makeVEFS`, `makeEVone`, `makeEVone`, `makeEF`. This can be generalized in an obvious way to faces of arbitrary degree simply by using a longer point array in line 4. Similarly the extrusion loop can be applied to higher degree faces by providing more points in the array in line 10; but note that the first point inserted in the extrusion, `:v7` in line 8, corresponds to the last point of the double-sided face, `:v3`.

A third loop for the hat. In lines 17-18 a halfedge and a point are pushed on the stack as parameters for the following lines 21-27. They show the typical constituents of a function: The parameters are popped from the stack and stored in registers. The last four images in the bottom row show how the hat is created by a single `makeEV` followed by a number of `makeEF`, one fewer than the face degree.

Note that the code for the hat does not need further generalization. It can be applied to faces of any degree as it is. The hat consists only of triangles, and to find the next triangle vertex it is sufficient to apply two `faceCCW` to an edge on the center vertex (line 24). The loop terminates only if this edge equals the first edge inserted with `makeEVone` (line 25).

Exit as an exception mechanism. The hat code does not use a fixed number of iterations. Instead it uses the loop-operator that continues to execute the body until the `exit`-operator is explicitly called. It (prematurely) terminates the innermost loop of any looping operator (`loop`, `repeat`, `for`, `forall` etc.). The program continues with the token *after* the respective looping operator. If `exit` is called outside all loops it terminates the program.

The `exit` operator illustrates again that advanced OOP concepts can be easily emulated in the GML, namely *exceptions*:

```
101 1 { process condition { 202 exit } if } repeat 202 eq { handle-exception } if
```

5.3.4 Example Arch or Door

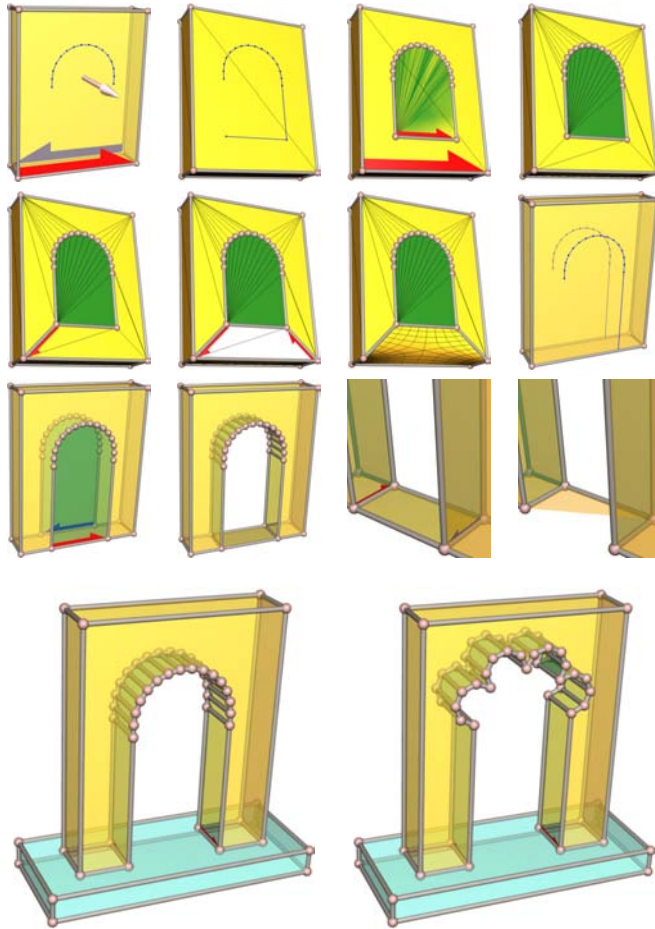


Figure 5.15: Creating a re-usable arch or door.

```

1  usereg !nrml !backwall !wall !poly
2
3  { usereg !door !wall
4    :door edgemate :wall killFmakeRH
5    :door edgemate faceCCW
6    :wall makeEkillR
7    dup faceCCW faceCCW
8    :door edgemate
9    exch makeEF pop
10   faceCCW killEF
11 } !glue-ringface-edges
12
13 :poly 0 get          !pr
14 :poly -1 get        !pl
15 :wall              vertexpos !pw0
16 :wall edgemate vertexpos !pw1
17 :pr :pw0 :pw1 project_ptline !prb
18 :pl :pw0 :pw1 project_ptline !plb
19 [ :plb :plb :prb :prb ]
20 :poly arrayappend   !poly
21
22 :poly :nrml neg :backwall faceplane
23 project_polyplane
24   5 poly2doubleface edgemate !backdoor
25 :poly 5 poly2doubleface !door
26 :wall :door :glue-ringface-edges
27 :backwall :backdoor :glue-ringface-edges
28 :backdoor faceCCW :door 2 bridgerings
29
30 !doorL
31 :doorL edgemate 2 faceCCW edgemate !doorR
32 :doorL edgemate faceCCW killEF
33 :doorR edgemate faceCCW killEmakeR pop
34 :doorL edgemate isBaseface {
35   :doorR edgemate makeFkillRH
36 } if
37
38 :doorL :doorR

```

Figure 5.16: A door in a wall as a function.

Consequently pursuing the path from the previous section, this section presents an example of a re-usable tool rather than a parametric model. Accordingly the code is formulated as a function rather than as a model. The startup lines are missing, instead the typical function header suggests that four parameters are expected on the stack: one polygon, two edges of the front- and backsides of a wall, and a vector that will in most cases be the normal of the wall, as further explained below. The four input parameters are shown in the first picture, Fig. 5.15 (1a).

Gluing a ring to its baseface when two of their edges coincide. When the front side opening is inserted the artist discovers that the opening for the back side requires exactly the same modeling steps again. As dictated by laziness he simply puts a pair of curly brackets around the modeling steps just performed and makes them a function `glue-ringface-edges`, shown in Fig. 5.16, lines 3-11. Note that the function is stored in a register, so it is a local function. As such it might use the same register frame as the function. But then it may never be executed *after* the finish of the door function, or an error will result. But it is also legal, and safer, that a register function creates its own new register frame with `usereg`. This also prepares the local function for becoming a modeling tool of its own.

The `glue-ringface-edges` function proceeds as follows. Its input parameters are the wall and the frontside of a double-sided face. Geometrically, both are usually part of the same plane, which is also the reason for the *z*-buffer artifacts in 5.15 (1c). The `killFmakeRH` operator in line 4 makes the backside of the double-sided face a ring of the wall (5.15, 1d), which removes the artifacts. The ring is then attached to the face boundary (Figs. 5.15, image 2a and 5.16, line 6) so that the wall face has again only a single boundary. With another edge the wall face is split (2b and line 9). Finally a `killEF` joins the newly inserted face with the bottom of the wall (2c and line 10), thereby removing the new face. It is important to note that the original lower boundary edge of the wall is destroyed by `glue-ringface-edges`. So it may not be used for modeling any more since when an operator attempts to dereference the token the interpreter stops with a B-rep error.

Completing the polygons for the openings. To create a doorway the two edges, the bottom of the wall and the bottom of the door opening, should be geometrically part of the same line. This is achieved by projecting both the start- and the end points :pr, :pl of the line strip :poly onto the line through the end points :pw0, :pw1 of the wall bottom edge (lines 13-18). Note that this yields vertical sides only when the wall bottom is horizontal. In case the ground has a slope a line intersection operator should be used instead. The points on the ground are then the intersection points of two vertical rays that emanate from :pr and :pl in downward direction, and the line along the wall bottom. But then also provision must be taken for the case when these rays do not both properly intersect the ground line.

An array is created from the two intersection points :plb, :prb, in this order (left first) to maintain a CCW orientation, because :poly is appended to it (line 20). Of course the intersection points could also be appended to :poly. But :poly is a function parameter (line 1), and array manipulation is always by reference; so the array of the *calling* function is affected, a behaviour that should be avoided whenever possible to prevent *side effects*.

Inserting the openings and connecting them. The new :poly is projected in direction :nrml on the back side of the wall (lines 22-23 and 2d). Then both polys are inserted into their respective walls using the local function :glue-ringface-edges discussed before (lines 26-27 and 3a). So in total two projection operators from the GML Geometry library are used in this example:

```

p p0 p1 project_ptline → q returns the point q on the line through points p0, p1 that is closest to p.
[ P ] dir n d project_polyplane → [ Q ] creates a copy Q of polygon P by projecting it in direction dir onto the
plane (n,d) with normal vector n and distance d.

```

Note that the bottom points :plb, :prb are duplicated when creating the array in line 19. The reason is that mode 5 of the poly2doubleface operator sets the temporary vertex flags according to the multiplicity of the points in the array (see section 4.5.1). This way the double-sided face has corners in the right places, e.g., at the bottom. The temporary vertex flags are then evaluated by mode 2 of the bridgerings operator to appropriately set the sharpness flags of the edges connecting the front- and backside openings (line 28 and 3b).

Removing the superfluous floor. The result of e0 e1 mode bridgerings is a halfedge that is uniquely determined: It is the newly inserted edge that connects the vertices of e0 and e1, and it is on the same vertex as e0. This halfedge is stored in :doorL in line 30, and it is shown as the left of the two halfedges in image (3c); the right one is :doorR that is obtained in line 31. To create an actual door the two sides of the wall must be separated at the bottom. This can be easily done. The faces of the two-sided floor in (3c) are joined with killEF (line 32), resulting in the situation in (3d): A single edge with the same face on both sides, which can only be removed with killEmakeR. This turns the face below the right side (:doorR edgemate) into a ring of the face below the left side of the wall (:doorL edgemate). Only in case the latter is not a ring itself (of yet another face) but a baseface, the ring :doorR edgemate just created must also be made a baseface (lines 34-36). Note that rings can have no further rings, but all rings that belong to the same baseface are equivalent.

The case that the face on the bottom of a wall is a ring is shown in the last two images in 5.15, (4a) and (4b). This happens when the wall is set onto a ground face. These two pictures also demonstrate the versatility of the modeling tool just created: A manifold of different arches and doorways can now be created very easily. As few as four powerful high-level parameters permit to create openings of any desired 2D shape. The tool can even cope with walls whose front- and backsides are *not parallel* by utilizing the :nrml parameter.

Parameter conditions for ‘consistent’ results. The parameter polygon :poly must lie in the plane of the front wall, and it must be oriented leftwards (first point right, last point left) because it represents the upper part of the opening. Furthermore, the size and orientations of the front- and back face planes must be compatible to the direction of projection, to assure that the projected polygon lies indeed within the back face. This can be critical at the bottom. When the ground plane is slanted it may be that :plb and :prb, when projected to the back, are *not* part of the line along the halfedge :backwall. – Finally, as mentioned, the ground plane of the wall must be horizontal to assure that the sides of the opening are vertical, because the sides are perpendicular to the ground plane.

Whenever creating re-usable tools it is vital to examine which conditions the parameters must fulfill to guarantee non-degenerate results. So under which conditions can the new tool be used? Which limitations exist? This involves a rigorous analysis of the valid parameter ranges as well as to identify the pre- and post-conditions of the tool.

Formal correctness means only that no GML errors occur and that the resulting mesh is geometrically consistent. But this is only a part of the correct application of a modeling tool. Much harder to specify precisely is the *intended purpose* of a tool: When can a tool be *reasonably* applied? – And sometimes creativity reveals that a tool is much more versatile than thought at first, and that it can be used for other purposes than it was conceived for. Remember that intermediate meshes may very well be geometrically inconsistent, and the end result may still be perfectly consistent (also see the discussion of consistency in sections 2.3, particularly 2.3.5, and 4.4.3).

5.4 Gothic Architecture



The main idea behind generative modeling is to describe the way how an object is built, rather than to describe only the result of the construction process.

The worsts case for generative modeling is when the surface of a three-dimensional object exhibits no regularities at all. If there is no identifiable relation between the different parts of the surface then the surface can only be considered more or less *random*. The only way to capture the shape is to sample it, i.e., to approximate it using geometric primitives (points, triangles, radial basis functions etc.). Note that a random surface is also the worst case for other surface compression schemes. Methods such as *delta encoding* for instance exploit the fact that a sample of a smooth surface can be derived from other nearby samples, e.g., by locally fitting a smooth curved or linear surface. The amount of *additional* information from each individual sample is only small, and it can be represented with only a few bits [IG03]. – Note that this in fact means that the surface can be locally described using a *reconstruction rule*, i.e., a generating function.

The best case for generative modeling is when a surface exhibits regularities, but these regularities are hard to capture using only a static set of rules, i.e., with something like delta encoding or recursive subdivision. A generative shape description consists of *dynamic rules* in the sense that geometric properties such as angles, directions, or distances can be measured, and they can trigger conditional branches in the rule sequence. Loops and conditionals permit arbitrarily complex, ‘dynamic’, behaviour within the rules. – But note that not only the rules are dynamic, but also the set of these rules is dynamic.

The outstanding feature of generative modeling is that it permits to re-combine existing rules to create new rules. This is a little bit like playing LEGO with rules and not with objects: All LEGO pieces have small cylinders on their top and little holes below, for the cylinders of other pieces to fit in. Quite similarly the GML permits to plug rules into other rules if they are compatible: GML operators can be concatenated if they have compatible signatures, which means that the output of one function can be used as input to the next.

The rule set of Gothic architecture. Gothic architecture, and especially the characteristic decoration of the windows, the *window tracery*, exhibits quite complex geometric shape configurations. But this complexity is achieved by combining only a few basic geometric patterns in ever-varying ways. This makes it an amazingly versatile but also quite challenging domain for parametric and procedural shape design. The Gothic *shape vocabulary* is so rich that for each and every church it can generate a unique set of windows, arches and column, which have almost certainly no duplicate somewhere else in the world. But still everything fits together seamlessly.

Gothic architecture has the very interesting property that, to some extent, it is the result of an optimization process. Cathedrals have been built long before the advent of exact methods to calculate masses and loads on walls, columns, and arches. The medieval builders have therefore used an intuitive trial-and-error method, and they have pushed the new technology to its limits. Especially in the first decades after 1140 many of the churches built in the new style were unstable and have collapsed. From these painful experiences the builders have learned, and they have developed sets of rules for cathedrals to keep standing upright for centuries. The rules are proportions that relate certain dimensions of the building to others. For instance there are rules for the thickness of a wall with respect to its height. But these rules typically take also the width of the wall into account, i.e., the distance of the supporting pillars to its right and left, and also the dimensions of these pillars. The pillars can not be chosen arbitrarily either; they are determined by the absolute height of the church nave, by the relation of the width to the height of the nave, and by the roof type. As a result, only an amazingly small number of high-level parameters can be freely chosen, they determine most of the other dimensions of a cathedral by applying the appropriate rules.

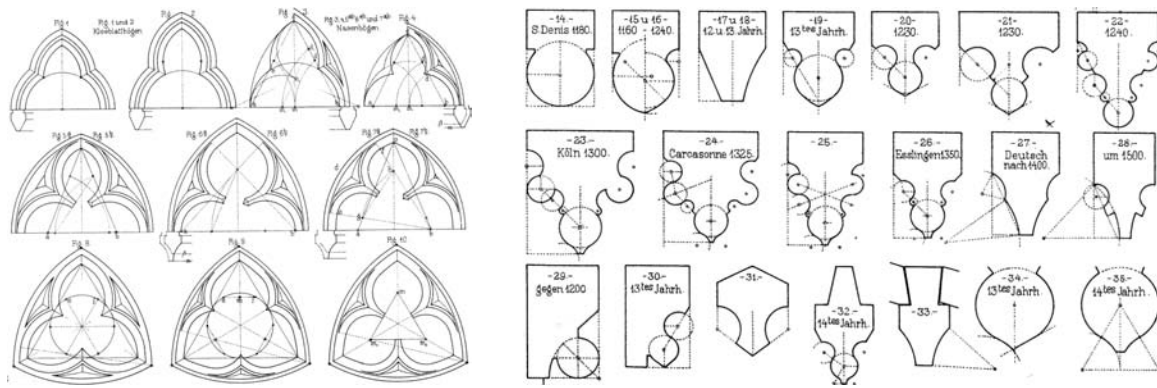


Figure 5.17: Constructions with compass and ruler. The profiles to the right are swept along the curves to the left.

Gothic construction rules are not public. In medieval times the Gothic proportions and construction rules have never been published, and they were rarely ever noted down. They were usually given from father to son only, or communicated only gradually within the building groups with their firm hierarchies. This was an early, and quite efficient, form of protecting the ‘know-how’ of the builders. Still today the commonly accessible knowledge is very limited, and after all the centuries there is still a rivalry among stone masons today, who perpetuate this form of knowledge protection¹.

To some (limited) extent, reverse-engineering is possible from drawings in older publications. Prof. Harmen Thies, head of the institute for historic building in the faculty of architecture at the TU Braunschweig, has provided invaluable advice for understanding Gothic architecture. The advent of the neo-Gothic style at the end of the 19th century gave rise to a number of books on Gothic proportions and styles. The drawings in Fig. 5.17 were published by von Egle and Fiechter in 1905, who show a great variety of examples of the basic constructions with compass and ruler [EF05].

The most comprehensive book with Gothic constructions and the most precise descriptions we have found so far is the *Lehrbuch der Gotischen Konstruktionen* from Georg Gottlob Ungewitter (1858), especially its famous fourth edition from 1904 edited by Karl Mohrmann [UM04]. It appears that the only available original medieval sources (in German) are six independent small little booklets, the *Werkmeisterbücher*. They are a written heritage from builders to their sons, such as for instance the famous *Büchlein von der Fialen Gerechtigkeit* from Matthias Roriczer (1486) [Ror86]. All six are presented by Ulrich Coenen in his PhD dissertation who edited, translated, and compared them faithfully [Coe88].

Modern publications on Gothic architecture are mostly descriptive and focus on classification and comparison rather than the actual execution of a building. Books such as *Masswerk* from Binding² [Bin89, Bin02] and also the *Bauformenlehre* from Koch [Koc00] give a good overview about generic parts and the development of the styles. Typical and generally applicable constructions can also be found in the educational literature for learning stone masonry [Ber96].

Problems when scanning a cathedral. The original motivation for applying the GML to Gothic architecture was to assess the feasibility of the new shape representation method with a non-trivial example domain.

Another more pragmatic incentive comes from the fact that Gothic architecture is a nightmare for automatic shape acquisition. The images in this chapter give only a rough idea of how difficult it is to capture a Gothic cathedral with a laser-range scanner. A central idea of Gothic architecture is to *dissolve* the static supporting stone structures to remove all the massiveness from the stone. An overwhelming wealth of small-scale detail is used to camouflage the walls. Larger plane surfaces are avoided whenever possible, and nearly every visible surface is decorated. The main problems are:

- **Small-scale detail everywhere**

Small-scale ornamental detail at the centimeter level can potentially appear everywhere. A serious constraint for the complexity of the decoration is that every part of the surface has to be accessible to the chisel (cf. Fig. 5.21). So a high sampling density is mandatory over the whole stone surface. Unfortunately, a cathedral can be very large.

- **Occlusion problem**

Floral decorations exhibit a surprising amount of three-dimensionality (see Fig. 5.20 (2a-c)). Frieases, columns, and vaults are spatial structures covered with decorating ornaments. So just a small portion of the surface is visible from any given point of view, and scanning has to be done from multiple viewpoints.

In summary, a cathedral would have to be scanned at very high resolution (millimeter) from a multitude of viewpoints. Given the typical size of a cathedral this makes scanning just impractical (see Fig. 5.44).

¹personal communication with two masters from the stone masonry school in Königsutter, one of four such institutions in Germany

²thanks to Dr. Thorhauer for this hint



Figure 5.18: The Braunschweig city hall. Building in neo-Gothic style from 1888.

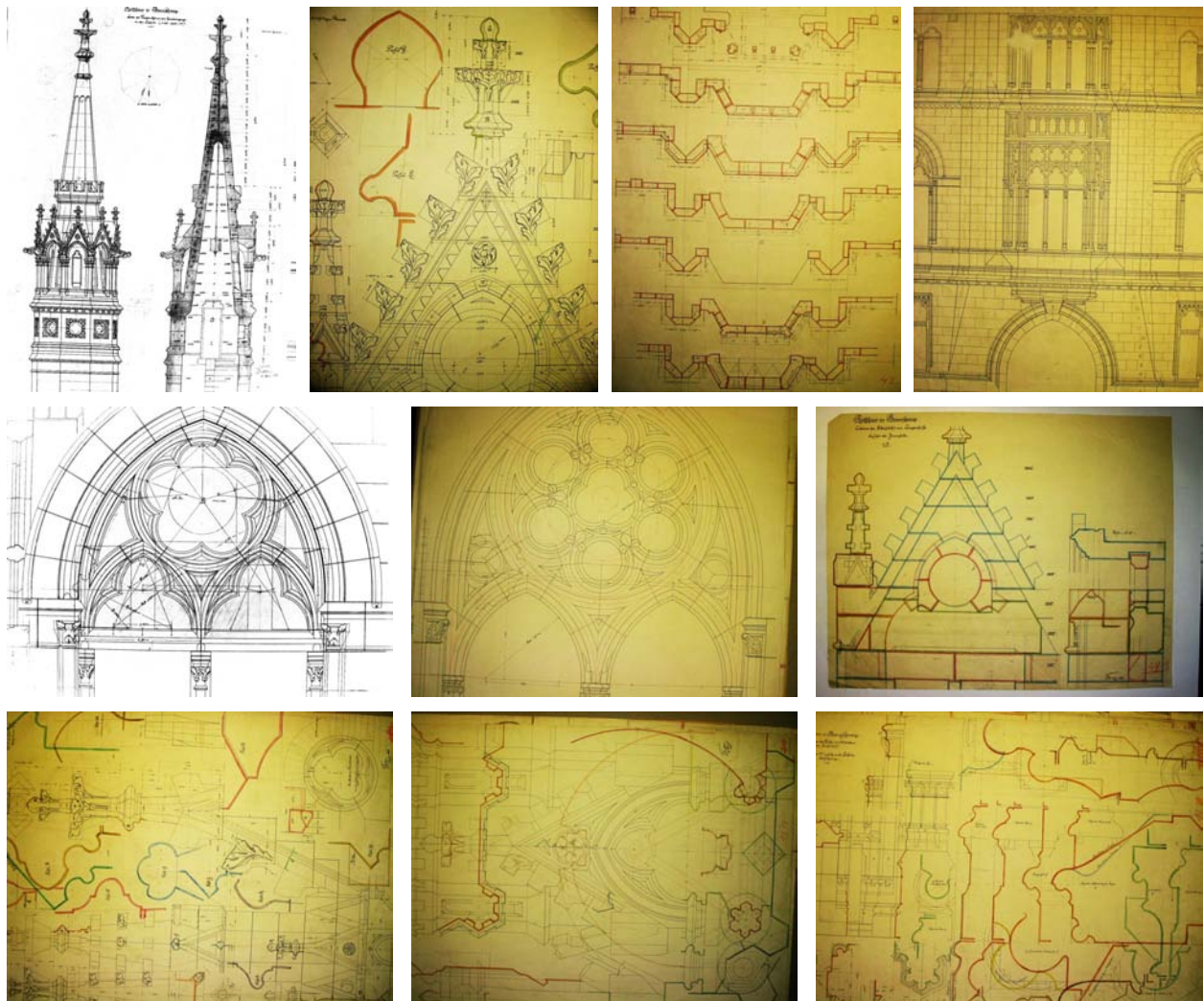


Figure 5.19: Highly coded construction plans from the Braunschweig city hall (from Hochbauamt Braunschweig)



Figure 5.20: Style and structure. Gothic vaults are based on a strict set of rules (1a-c). Even floral ornaments with their abundance of free-form detail exhibit a strict regularity (2a-c). The similarity of different windows becomes apparent when seen from inside (1d,2d). Interior fotos from inside the Braunschweig city hall.



Figure 5.21: For centuries, highly coded construction plans were turned into precise pieces of stone by skilled stone masons. Only through his experience a stone mason can turn a plan like in Fig. 5.19 into a detailed sequence of construction steps. Row 1: The plan is re-constructed in 1:1 scale directly on the stone. Row 2: Plane surfaces and right angles are kept as long as possible, to enable precise measurements. The planes form as tight as possible bounding boxes for the free-form ornaments. Row 3: The final precision is so high that the boundary between different stones is hard to tell. – Images of stone masons at work at the Stephan's cathedral in Vienna, from the *Digital European Cathedral Archive* at www.deca-forum.net.

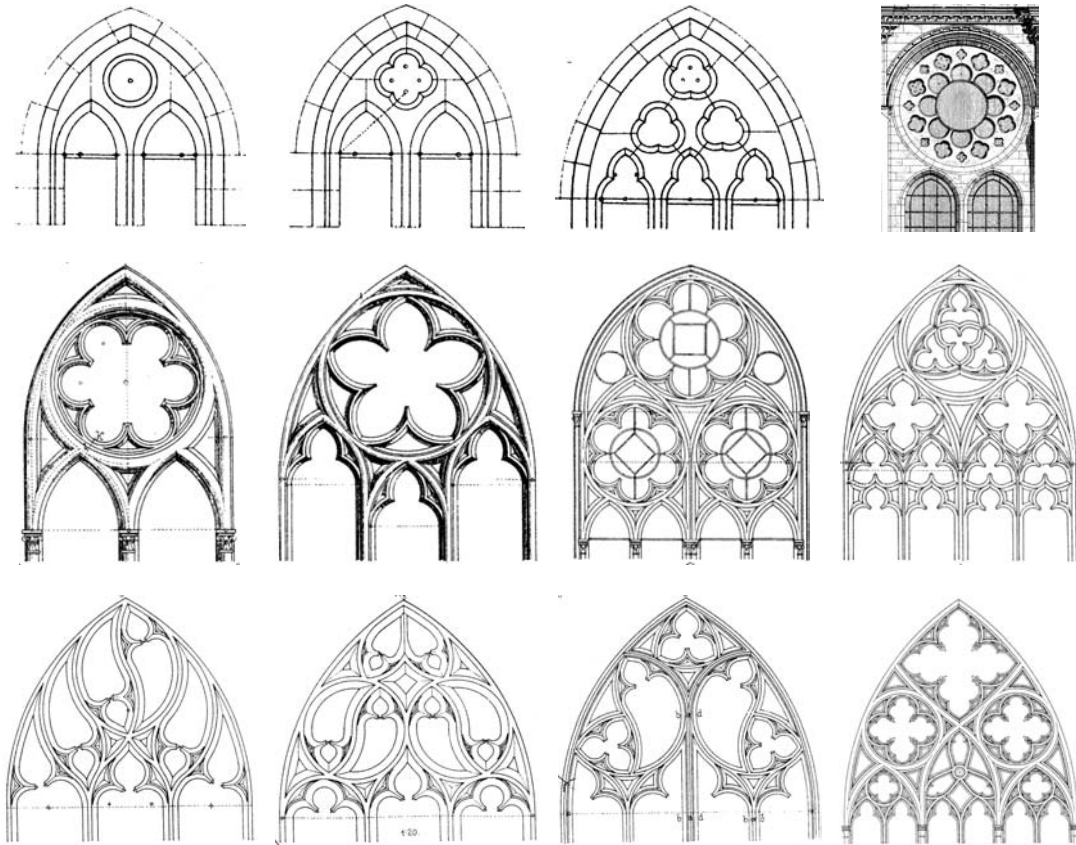


Figure 5.22: Evolution of Gothic window tracery. Row 1: Bar tracery in the early Gothic period (starting 1140), Row 2: Evolving standard window patterns and style vocabulary in the high Gothic time (from around 1250 on), and Row 3: Flamboyant richness and recursive patterns in the late Gothic period in the 14th and 15th centuries. Drawings are from Egle [EF05].

5.4.1 The construction of a Gothic Window

Window tracery is the very particular and characteristic type of window decoration found in any building of Gothic style. Gothic architecture, and especially window tracery, exhibits quite complex geometric shape configurations. But this complexity is achieved by combining only a few basic geometric patterns, namely circles and straight lines. They are combined and obtained from each other by a limited set of operations, such as intersection, offsetting, and extrusions. The reason for this lies in the nature of the process how these objects have been physically realized, i.e., through constructions with compass and ruler. *All free-form curves used in Gothic architecture are exclusively made of circle segments.* The reason is the *scaling problem*: A shape drawn on a piece of paper must somehow be reproduced in 1:1 scale in order to build the object. How could an arbitrary shape drawn on paper be extremely enlarged? This was only possible when the drawing could in fact be re-constructed, which made the use of ruler and compass mandatory.

Every single stone of the building has to appear somewhere on a plan, because each stone is individually manufactured by a stone mason. The traditional way to communicate the construction, e.g., of a particular window, is by a series of drawings. Over the centuries, these drawings have evolved into a very domain specific, condensed code (as in Fig. 5.19), which was essentially a compressed communication form between the builder (architect) and the stone masons. The construction process itself, however, has only been based on extensive experience, cf. Fig. 5.21. It has never been formalized in an unambiguous way so that, e.g., a computer could reproduce results of equal quality.

To do so requires two fundamental steps:

- **Analysis:** to identify the basic operations used in window construction and decoration, and
- **Synthesis:** to express the parameterized constructions formally and unambiguously using the GML.

Both the analysis and the synthesis were described in a short paper on the Solid Modeling conference 2004 in Genua, and in extended form on the VAST 2004 conference in Brussels [HF04a, HF04b].

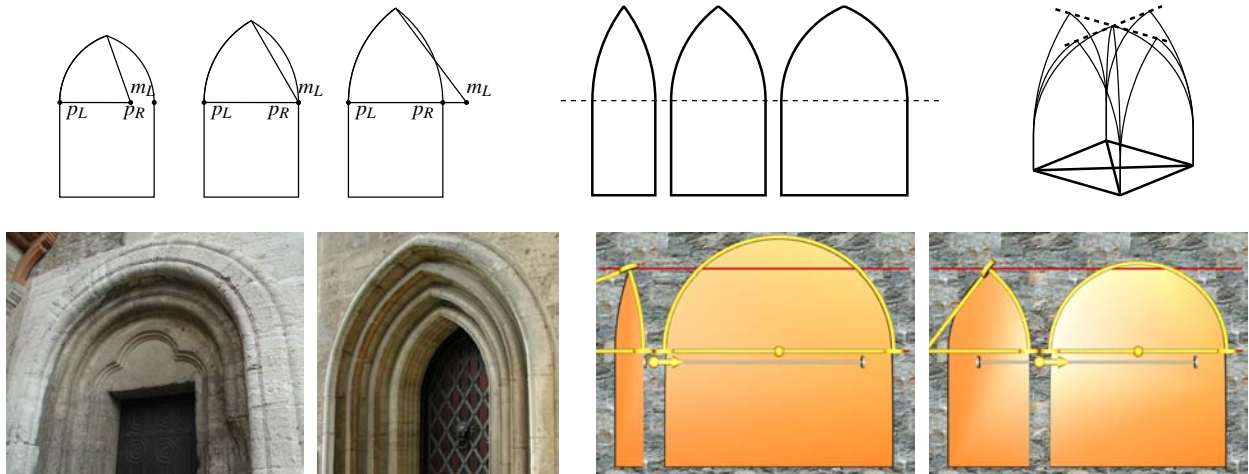


Figure 5.23: (1a-c): Gothic arch with varying excess: Four-centered (0.75), equilateral (1.0), and pointed arch (1.25). (1d): The height of a pointed arch can be kept constant even when its width varies. (1e): The crossing of two pointed vaults can be realized with a pair of pointed arches. (2a,b): A round arch can be offset by scaling, but the offset of a pointed arch has a different excess. (2c,d): The height of a round arch is always half of its width



Figure 5.24: International success: Stephan's Cathedral in Vienna (a), Cologne Cathedral (b), and the Cathedral in Braunschweig (c,d).

The pointed arch. This is the most distinct basic pattern in Gothic Architecture. Its geometric construction is based on the intersection of two circles. The circles are tangent continuous to the sides of an arch or a window, given as two vertical line segments (Fig. 5.23). Consequently the midpoints m_L and m_R of the left and right circle segments lie on the horizontal line through the upper endpoints p_L and p_R of the left and right segments, the *arch basis points*. The pointed arch is symmetric, so both circles have the same radius $r = \text{dist}(p_L, m_R) = \text{dist}(p_R, m_L)$.

The ratio $r/\text{dist}(p_L, p_R)$ can be called the *excess* of the arch. When the excess is 1.0, the circle midpoints coincide with the upper segment endpoints. Together with the circle intersection, they form an equilateral triangle. This is the standard pointed arch, also called the *equilateral arch*. With an excess > 1.0 , the circles intersect at a sharper angle, and this is what is actually called a *pointed arch*. When the excess is < 1.0 , the arch is not so high, and this is called *four-centered arch*. The extreme case is the round arch with an excess of 0.5, so that m_L, m_R coincide in the midpoint between p_L, p_R .

The historical development of window tracery. The pointed arch was a technological breakthrough that, after its introduction around 1140, has truly revolutionized the construction of cathedrals. It is a generalization of its predecessor, the *round arch*. The pointed arch was first systematically employed by abbot Suger in the renewal of the cathedral of St. Denis (near Paris, France), and the new style spread over all Europe in just a few decades. It has dominated the European sacral architecture for more than two hundred years, and gave rise to a veritable footrace between cities, with cathedrals becoming ever more sophisticated and risky, and tolerances becoming smaller and smaller. The overwhelming success of the Gothic architecture is witnessed by the great similarity of churches from different countries (Fig. 5.24).

Technologically, the great advantage of the pointed arch over the round arch is the fact that the distance between the columns could now be varied without affecting the height of the arch (Fig. 5.23 (d)). This leaves greater much flexibility for positioning the columns, and it helps to solve delicate problems with the design of the ground layout in a cathedral, especially around the chorus (see Fig. 5.45 (3a)).

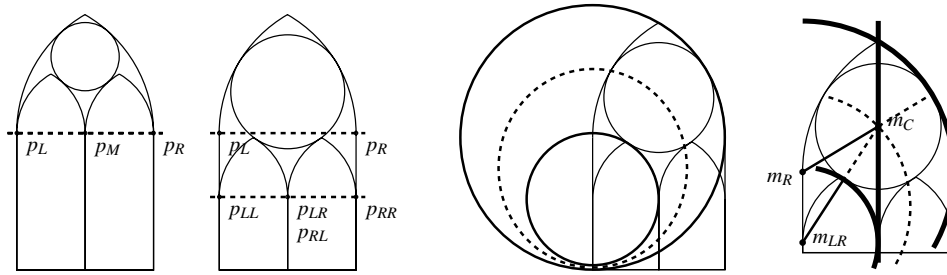


Figure 5.25: Geometry of the prototype window. (a,b). Calculating midpoint and radius of the rosette's circle (c,d). Points on the dotted line in (c) have the same distance to both circles. As revealed by (d), it is an ellipse.

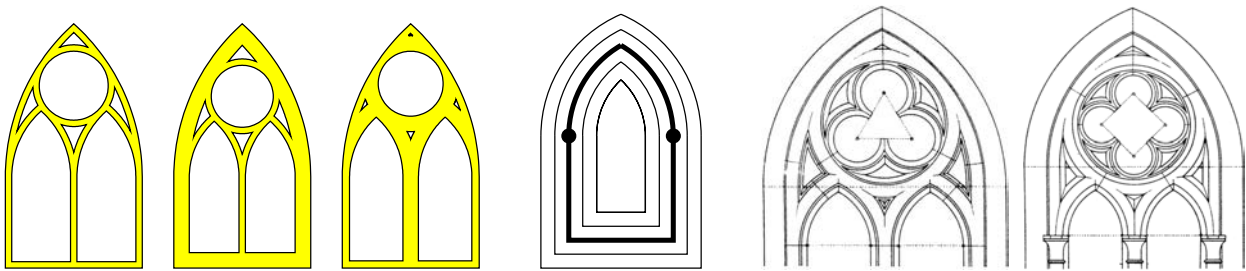


Figure 5.26: The fields within a Gothic window. (a,b,c): The seven different fields from Fig. 5.25 (a,b) are shrunk to embed them in a single border plane. (d): The offset operation changes the excess of a pointed arch, but keeps the circle midpoints constant. (e,f): Examples for field boundaries in historic drawings from Egle [EF05]. The rosette circles are filled with a lying trefoil and a standing quatrefoil. Compare to Figs. 5.17 (3a-c) and 5.29.

Let there be light. Basically the same shape as for an arch can also be used for a window. The idea of Gothic cathedrals is to make the walls of the church as transparent as possible, in order to let a maximum of light enter the room. With coloured windows, a cathedral was flooded with light in all colors, which was one of the manifestations of God in the perception of the medieval christian. The size of the windows in relation to the size of the walls increased, and the walls actually “dissolved” to the point where they completely lost their supporting function. Gothic cathedrals get their stability almost exclusively from columns, and not from walls [Bin02].

There is a remarkable development of the ornamental decoration in the upper part of the window, the *couronnement* (Fig. 5.22). In the Early Gothic period, starting around 1140, the windows were created by cutting openings into large stone plates in the wall. This premature form of window tracery is therefore called *plate tracery*. In the High Gothic period, from around 1250 on, the stone parts became ever thinner, and the windows covered an increasing portion of the wall. The glass windows were set into a network of individual stones, the *bar tracery* (5.22, row 2). The late Gothic period, in the 14th and 15th centuries, saw a great refinement and sophistication of window tracery. The basic patterns were varied over and over again, with recursive sub-structures and self-similarity, to the point where the static stone appeared to be actually flowing. An example is the French and English *flamboyant* style (5.22, row 3) with its flame symbolics [Bin89].

The prototype window. A very common and basic High Gothic window type is one with two sub-windows that are also pointed arches, as for example the window in Fig. 5.22 (2a). It exhibits the main shape features, the *shape vocabulary*, that was subsequently refined and varied in the Late Gothic period. The sub-windows have most often the same excess as the main arch, which makes the excess a high-level parameter of the window.

So this window type was chosen as the prototype window for reconstruction, and it is shown in Fig. 5.25. The window in (a) has excess 1, so that the midpoints of the circular arcs and the basis points p_L , p_R coincide. They are also basis points of the subwindow arches. But like in Fig. 5.22 (2a) it is often the case that the sub-arches are set down with respect to the big arch, in order to create a larger space for the couronnement. So the vertical distance between the basis points of the outer and inner arches is another high-level parameter of the window (Fig. 5.25 (b)).

Adding a circular rosette. The space between the outer and inner pointed arches can be filled in many different ways. This decoration is the distinguishing feature of each individual window. In the early days of Gothic, this space was quite often filled with a circular rosette. Geometrically the problem is to find the midpoint m_C and radius r_C of a circle that

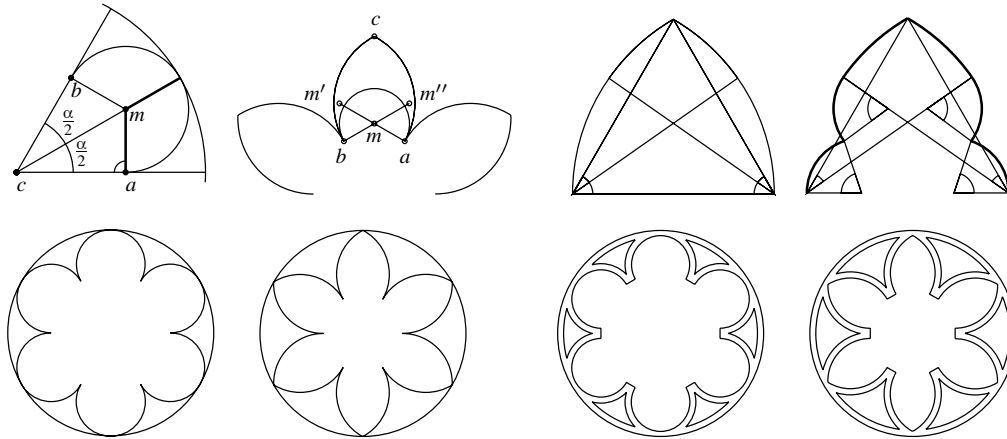


Figure 5.27: Construction of rosettes with round foils and pointed foils. (1a): Computing the arc of a rosette with six rounded foils, so $\alpha = \frac{2\pi}{6}$. (1b): Relative displacement of m of 1.15 creating midpoints m' and m'' to obtain pointed foils. (1c,1d): Pointed trefoil arch obtained from pointed arch by replacing part of the arcs. (2a,b): The circle segments partition the rosette into fields which are then shrunk (2b,c) like the window fields in Fig. 5.26

touches the outer arch and two segments of the inner arches. Consider the set of all points in between a big arc and a sub-arc, for instance arc_{LR} (left sub-window, right arc) and arc_R (big arch, right arc), as in Fig. 5.25 (c). The points that have the same distance to both respective circles, (m_R, r_R) and (m_{LR}, r_{LR}) , are shown as dotted curve.

This curve is an ellipse, which is revealed in Fig. 5.25 (d): Connect a point on it, for instance m_C , with both midpoints m_R and m_{LR} . The distance from m_C to the upper midpoint m_R is less than the radius r_R of the big circle (dotted continuation), so $\text{dist}(m_C, m_R) = r_R - x$ for some $x > 0$. Similarly, the distance from m_C to the lower midpoint is $\text{dist}(m_C, m_{LR}) = r_{LR} + y$ for some $y > 0$. But m_C has the same distance to both circles, so $x = y$. Then x cancels out from the sum of both distances, and $\text{dist}(m_C, m_R) + \text{dist}(m_C, m_{LR}) = r_R - x + r_{LR} + x = r_R + r_{LR}$ is constant. Since this holds for all points on the dotted curve, it must be an ellipse, and m_R and m_{LR} its foci. The midpoint m_C of the rosette can then be obtained as the intersection of this ellipse $(m_R, m_{LR}, r_R + r_{LR})$ with the vertical axis of symmetry. Practically it can be computed by intersecting a unit circle with an affinely transformed line.

Offset curves. The rosette circle and the sub-arches partition the window into disjoint regions or fields. These fields define the basic structure of the window, which is then further refined. This can be done by simply adding a profile around the actual window holes to emphasize the shape, or, especially in the later period, by adding sub-structures, again composed of lines and circular arcs. It is very common that there is a thin planar border between adjacent fields, so that there is actually a single connected border plane. Geometrically this means that the field border is offset by a certain distance, as depicted in Fig. 5.26 (a,b), so that one contiguous border plane results (shown in yellow). Regarding the great variety of examples where this pattern is used, it is reasonable to distinguish between two different offset parameters:

- the interior offset distance of the fields from each other, and
- the offset distance from the ensemble to the outer pointed arch

Both parameters are equal in Fig. 5.26 (a), while in 5.26 (b) the outer offset is doubled, and in 5.26 (c) the interior offset.

One great thing about the circle is that its offset is again a circle. This applies also to curves that are created from a sequence of circular arcs and line segments, like for instance a pointed arch. But note that if the sequence contains corners, e.g., two arcs joining in an intersection of the respective circles, the intersection of the offset circles must be computed for obtaining the offset curve sequence. Simple scaling is not sufficient to create offset curves: The offset of a pointed arch has a different excess than the original arch, as it is shown in Fig. 5.26 (d), and also before in Fig. 5.23 (2b).

Rosette window with multiple foils. A very common way to fill a circular field is by a rosette with multiple foils, for example a trefoil or a quatrefoil. The foils come in a variety of different shapes; common variants are round and pointed foils (Fig. 5.27 (1a,b)). A further distinction is between *lying* and *standing* rosettes, shown in Fig. 5.29 (1a,1b) and (2a,2b).

The geometry is fairly straightforward: Given the number n of foils in a unit circle, the radius r of the round foils is computed as in Fig. 5.27 (2a). Consider the tangent from center c to the circle (m, r) . The distance from c to m is $1 - r$, so the length of the perpendicular from m to the tangent is $\text{dist}(m, a) = (1 - r) \sin \frac{\alpha}{2}$. But $\text{dist}(m, a) = \text{dist}(m, b)$ is also supposed to be r . This equation gives $r = \sin \frac{\alpha}{2} / (1 + \sin \frac{\alpha}{2})$. The perpendicular feet a and b are the endpoints of the circular arc that is rotated and copied n times to make up the rosette.



Figure 5.28: Window tracery with flat and round bars. With the flat style (a-c) the front sides of the bars form a contiguous front plane, of which the window and fillet fields are *rings*. This is opposed to the french style (d) with round bars. Compare the french style with the profiles created with stable extrusion in Fig. 4.80. Images are taken from the Old Town Hall (a) and the Martini church (b-d) in Braunschweig.

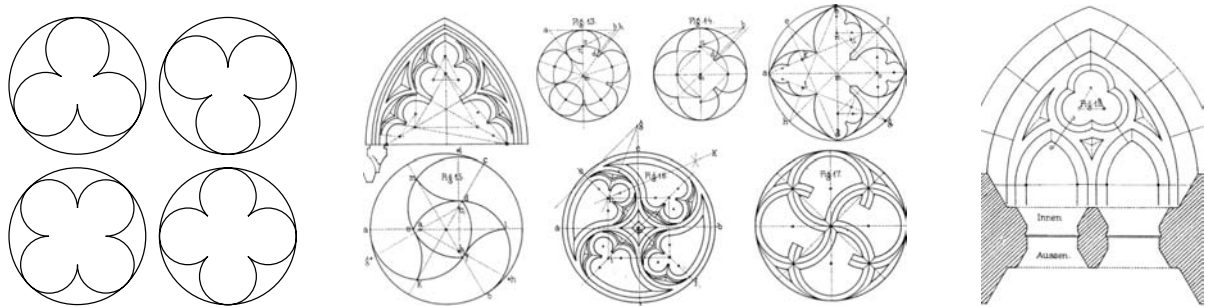


Figure 5.29: Basic and sophisticated rosettes. (1a,1b,2a,2b): Lying and standing trefoil and quatrefoil rosettes. Center: Sophisticated rosettes from the late Gothic period. Right: In illustrations the profile is often drawn right below the upright image, towards which it needs to be turned by 90 degrees. The drawings are from Egle [EF05].

The pointed foils are a variant of the round foils. Consequently they can be obtained from the round foils, as it is shown in Fig. 5.27 (2b): The midpoints m' , m'' are obtained from m by displacement along the lines (a, m) and (b, m) . The pointedness, and thus the radius of the circles, is influenced by the amount of displacement, which can be specified in relation to the original radius. Points a and b and the intersection point c then specify the arcs which make up the pointed foils. In order to fit into the original circle, the foils are simply scaled smaller.

Just as described in section 5.4.1, a connected boundary region for the rosette is constructed from the network of circular arcs. Examples are shown in Fig. 5.27 (1c,d). Note that also these offset curves also only consist of arcs and line segments.

Further refinements. Circular arcs can be combined very flexibly. Both corners and tangent continuous joints can be obtained from quite elementary geometric constructions. The pointed trefoil arch in Fig. 5.27 (2c,d) for instance is easily obtained from a pointed arch: First both arcs are symmetrically split, and then the lower parts are replaced each by a pair of smaller arcs. When joining circular arcs, tangent continuity is obtained simply by choosing the midpoint of the next arc on the line through mid- to endpoint of the given arc. This principle is the source of the great variability of geometric patterns in Gothic architecture. It is also the principle used in the drawings from Egle in Figs. 5.17 (3a-c) and 5.29.

Appearance: Profiles So far the structure of the window is solely defined by a few two-dimensional fields whose boundaries are composed of circular arcs and line segments. The fascinating and impressive three-dimensionality of Gothic windows is achieved by profiles that give depth to the two-dimensional geometric figures. In architectural illustrations, profiles can often be found above or below a front view, like in Fig. 5.29 (rightmost). – Sometimes, in more coded versions, profiles are also set into the drawing itself, for instance in the plans of the Braunschweig city hall in Fig. 5.19.

Technically, these profiles are swept along the field border curves, the profile plane being orthogonal to the tangent of the curve. At corner points, where the tangent is discontinuous, the sweep is basically continued onto the bisector plane. This is the plane that is spanned by the bisecting line of the angle in the corner and the normal of the 2D construction plane. Yet this is only the case when the curve is locally symmetric to the bisector plane. In more general cases, the discontinuity locally follows the *medial axis* of the two parts of the curve.

• excess	the excess of the main pointed arch and the sub-arches	Fig. 5.23 (a-c)
• arcDown	vertical offset of the base points of the sub-arches	Fig. 5.25 (b)
• bdOuter	offset distance of the fields from the main arch	Fig. 5.26 (a-c)
• bdInner	interior offset distance of fields from each other	Fig. 5.26 (a-c)
• wallSetback	distance the whole window is set back into the wall	
• heightBott	height of the horizontal bottom of the window above the ground	
• kseg	accuracy for circles and circle segments (<i>n</i> -gon)	
• Style	high-level style parameter specifying how to fill the seven (eight) fields	

Figure 5.30: The parameters of the standard Gothic window. All parameters are stored in a dictionary. The value of the Style parameter is another dictionary with the four style generating functions (examples in Figs. 5.32, 5.33).

5.4.2 The Gothic Window in GML

The analysis has revealed the elements a standard Gothic window is composed of. The realization as a GML model is a bit more complicated than the previous examples, but as a matter of fact it is quite straightforward.

The parameters of the window. A generative Gothic window is of course a configurable modeling tool, and using the GML it is represented as a function. The analysis has shown that this function can potentially have a large number of parameters. Some of the parameters are more generic than others: The overall shape is a pointed arch, which in any case has an excess parameter. Furthermore this first realization shall restrict itself to exactly two sub-windows and a circular rosette. With the prototype window from Fig. 5.25 the height offset between the base points becomes another parameter.

The window is partitioned into seven fields that are set into a common border plane, as shown in Fig. 5.26. When using combined B-reps as shape representation then a discrete control meshes is needed for the curved parts of the surface. So another parameter is required to specify approximation quality of the control mesh. All in all the eight parameters listed in Fig. 5.30 are needed to describe the basic shape of the window.

The larger number of parameters makes the usual parameter passing over the stack a bit inconvenient. Especially when sets of parameters are to be re-used, or slightly changed, it is more convenient to use a dictionary. The usage of the Gothic window modeling tool is therefore as depicted in Fig. 5.31: All values are entered in a dictionary which is then used as a function parameter (line 13). The concrete parameters in line 12 describe the particular wall the window is to be cut into, two halfedges and the base points of the arch. They should geometrically lie in the face plane of the wall.

Style libraries. The most intricate, and the most interesting, fundamental question is how to describe the **Style** of a particular window – especially in the light of the great variety of Gothic forms in the figures so far. This is again a variant of the shape description problem: All the possible variants can not be described using only a fixed set of concrete values. With a generative approach for shape description the solution is again to use parameterized functions, and again they are wrapped in a dictionary.

The value of the Style property used in the example from Fig. 5.31 is Gothic-Window.Styles.Style-1. This is a dictionary that contains four *style generating* functions, corresponding to the four types of fields in the window. They must have the following prescribed signatures:

```

poly b d eWall eBack      style-main-arch  →  eArch
    creates the main arch of the window from poly and returns it as a new face eArch; the distance between the
    face planes of eArch and eWall must be d, and b the offset from the inner fields to the outer arch

poly eArch eBack          style-fillet      →  -
    decoration of the four fillets, set into the main arch eArch created before, may create opening using eBack

poly eArch eBack m r      style-rosette   →  -
    sets the rosette into the main arch, may create opening; poly is the circle polygon of the rosette circle (m,r)

poly eArch eBack arcL bh  style-sub-arch  →  -
    realizes the two pointed sub-arches. The pointed arch polygon poly is set into the main arch. arcL is one of
    the two arcs from which poly was generated, and bh is the height of the bottom of the sub-arch

```

The definition of a very simple style with its four functions is shown in Fig. 5.32. All functions do basically the same, a simple extrusion in different colors. The next idea is to exploit this similarity to set up a library of re-usable profiles. Consequently pursuing this idea leads to a much more concise representation of much more complex styles.

An example is the ‘high-level style’ definition in Fig. 5.33, which directly routes the input parameters to prefabricated functions from the Profile library. In addition to profiles like profile-4 this library contains also utility functions, for instance rosette-pointed. This function is compatible to the the parameters of style-rosette listed above, but its working must be further configured with mode flags and a profile parameter that, by the use of the stack, can simply be pushed behind (5.33, lines 9-10).

Recursive styles. The most fascinating option is based on the fact that the sub-arches are pointed arches as well. This recursive structure of Gothic architecture can be directly represented in the GML with a *recursive style*. In this case the function Profiles.sub-arch-recursive expects another style dictionary, just like the one from Fig. 5.31. In Style-8 from Fig. 5.33 the style dictionary is assembled ‘on the fly’ by another utility function Profiles.make-style-dict.

A recursive style may even be *doubly recursive*; the definition of Style-7, which is used as sub-style in Style-8, looks very much like the definition of Style-8 itself. It is also recursive, except that it uses another style Style-4 as style for the sub-arches. So the recursion in Style-8 has in fact depth two, and the resulting window has $2 \cdot 2 \cdot 2 = 8$ sub-windows. Results with different styles are shown in Fig. 5.40 later.

Like with any other software library a vital prerequisite for a good style library is a good interface definition. This is also the intellectually most challenging part, since it requires to decide which information is exchanged between the modules. In case of the window styles, all four style generating functions obtain a polygon poly, i.e., a curve that is already sampled. The rosette and sub-arch functions though obtain higher level information in addition to the polygon, a circle and a circle segment (*circular arc*). They actually make the respective poly parameters redundant. But higher-level information is needed for all decorations that are more intricate than just a simple profile. To see why this is so it is necessary to know how circular arcs and pointed arches are actually represented in the GML.

Circle segments in the GML. A 2D plane in 3D can be parameterized in many ways. To minimize redundancy GML operators expect just a normal vector and the distance from the origin, i.e., a vector and a float, for a plane.

Several possibilities exist also to parameterize a circular arc: By midpoint and two polar angles, by three points on the arc (as in many drawing programs), by two points plus circle center etc. But to represent a circle segment in general position in 3-space, all parametrizations using 2D coordinates need a basis for the plane, i.e., two 3D unit vectors e_0 , e_1 that are perpendicular to each other. This is inconvenient and computationally not very stable: The parametrization should efficiently support the typical operations on circle segments, like affine 3D transformations (general translations, rotations), the offset operation, and the conversion to a circle.

Furthermore, since Gothic plans are made of both line and circle segments, their parametrizations should be compatible. The chosen parametrization of a circle segment is therefore like a line segment [a b] with start- and endpoint, plus the midpoint m of the circle (see Fig. 5.27 (2a)). In the GML, this is can be written as an array of three points [a m b]. To distinguish between the two possible arcs from a to b, a normal vector n is specified for the orientation as well, with the convention that the arc is always CCW oriented when the normal points to the viewer. Such a circle segment can the be converted to a polygon with the circleseg operator:

[a m b] n rml n mode **circleseg** → [p1 .. pk]

creates a polygon with evenly spaced points on a circular arc with radius dist(a,m) in the plane with n rml containing a. mode determines the meaning of n: In mode 0, the polygon is approximately part of an n-gon, in mode 1 it has n + 1 points. In mode 2 the length of the polygon segments is approximately n (as float)

What is ‘the pointed arch’? The interesting property of the pointed arch is that it exists on several different levels of abstraction. This is also symptomatic for the human perception of shape: ‘A shape’ means in most cases rather a whole class of shapes, with fuzzy class boundaries. Furthermore there are several different roads to understanding the same shape, and on several different levels. Each level deserves its own representation, and the transition from one level to another is of course achieved by a function.

- | | |
|---|---|
| (a) High-level parameters | pL pR excess offset heightBott n rml |
| (b) Closed curve from two circle- and three line segments | [pR mR pT] [pT mL pL] [pL bL] [bL bR] [bR pR] |
| (c) Closed polygon from sampling the curve | [...] |
| (d) Mesh face created from closed polygon | eArch |
| (e) Control mesh of a highly decorated window frame with ornaments and profiles | |
| (f) Tesselated high-resolution triangle mesh for display | |

The steps (a)-(c) are always the same, so they are realized with a ‘static’ function from the Gothic window library . The steps (d)-(e) are configurable, since they are determined by the choice for a style for the particular window, as in Fig. 5.31. Note that the style can also choose to redo (a)-(c) in a completely different way, and replace the simple pre-sampled polygon by another, more complex, shape. Steps (e) and (f) finally are taken care of by the combined B-reps.

Pointed arch: From (a) high-level parameters to (b) circle segments. The pointed arch as a conversion function `gw-pointed-arch` is listed in Fig. 5.37. The first line of code just retrieves the parameters from the stack (in reversed order). Line 3 computes the midpoint of the right arc $mR = pR + excess \cdot (pL - pR)$. Note that it lies to the left, since $excess \approx 1$. The radius `rad` of both circles is computed as the distance from `pR` to `mR` minus the desired offset; like with extrusion, positive offsets are inwards. The points where the two circles intersect are computed with the circle intersection operator; only one of them is needed. The circular arcs are then assembled inline. They are pushed in the order left, right so that they are ready to be processed right, left to produce a CCW polygon.

```

p0 p1 t line_2pt      → q    computes a point on the line  $q = p0 + t \cdot (p1 - p0) = (1 - t) \cdot p0 + t \cdot p1$ 
  p q t move_2pt     → p'    moves p into the direction of q by t absolute units.
m0 r0 m1 r1 nrml intersect_circles → p0 p1 computes the intersection points of the circles (m0,r0) and (m1,r1) in
                                         plane nrml. When nrml points to the viewer and m0 is left and m1 right,
                                         then p0 is below and p1 above the line segment [m0 m1].

```

Note that the order of the midpoints in line 7 of `gw-pointed-arch` (5.37) is reversed; `mR` is the midpoint of the right arc, which lies to the left. Therefore the second result of `intersect_circles` is popped, and not the first. – The family of pointed arches in Fig. 5.23 (a-c) was created using this function by varying the `excess` parameter. In Fig. 5.26 (a-c) the offset was varied; the base points are marked there with big dots.

Pointed arch: From (b) line and circle segments to (c) the polygon. Only the two arcs have been computed so far. To get also the end points `bL`, `bR` of the horizontal line at the bottom it is enough to specify the bottom height, `heightBott`. The points are then obtained by setting the end point of the left arc and the start point of the right arc to the bottom height. This does the function `gw-polygon-2arcs-height` listed in Fig. 5.38. An arc is an array of three points. With `:arcR 0 get :hb` putZ its first point is retrieved and the z coordinate of this point is set to `:hb` ('height bottom'). In line 3 an array is made of twice this point, so that the other polygons generated by `circleseg` in lines 4-5 can be simply concatenated to it. The left end point is also appended twice in line 6.

Note that the z coordinate is interpreted as 'height above the ground'. Especially for city modeling it is much more natural to make the (x,y) -plane the ground plane.

The resulting polygon contains exactly three points twice: The tip of the arch, which is the startpoint of one arc and the endpoint of the other, and the two points on the bottom. The reason is that the `poly2doubleface` operator (in mode 5) creates sharp corners only from all polygon points that appear successively more than once (see section 4.5.1).

The Gothic Window function. It was shown how a single pointed arch is processed across the different stages. This knowledge can now be applied to create a whole window.

The main window function is `gw-gothic-window` from the `Gothic-Window.Tools` library, listed in Fig. 5.36. It has two tasks: (a) to compute and polygonize of the eight fields making up the window, and (b) to call the style generating functions with the appropriate polygons. The eight fields are the main arch itself (`arcR`, `arcL`), the rosette (`rosetteMid`, `rosetteRad`), the right sub-arch (`arcRR`,`arcRL`), the left sub-arch (`arcLR`,`arcLL`), and the four fillets.

As explained before the function `gw-gothic-window` has five parameters: the halfedges on the front- and backsides of a wall, the left and right base points of the arch, lying in the frontside face plane, and the window dictionary. The latter contains additional parameters that are accessed via name lookup: The function starts by beginning the window dictionary (line 5). The next statement `Style begin` already retrieves the style entry from the window dictionary, and it adds the four window generating functions to the current modeling vocabulary. – Note that also *name overloading* is possible; another previously begun dictionary might for instance contain four default style functions, and the new dictionary overwrites only one or two of them. – The function can be divided into five parts overall:

- lines 18-21: **Decorate the main arch**

The main arch is processed via operator chaining: the base points and `excess` are sent as high-level parameters to `gw-pointed-arch` → `line` and circle segments plus bottom height to `gw-polygon-2arcs-height` → directly routed to the style generating function **style-main-arch**. This function is then supposed to produce the window front face `edgeArch` that is subsequently given to the other style functions.

- lines 25-28: **Compute the sub-arcs and the rosette**

The points `pL`, `pR` are moved into the wall by `wallSetback` to serve as displaced base points, the resulting displaced main arch is needed to compute the inner fields. The rosette circle, and consequently also the arcs of the sub-arcs, are computed by another function `gw-fillets-arcs-rosette` that is described in the next paragraph.

- lines 32-37: **Compute and decorate the four fillets**

Like the pointed arch the fillets are processed in two stages: First circle segments are computed (`gw-compute-fillets`), which are then polygonized (`gw-polygon-fillets`), to finally apply **style-fillet** to each of them. The conversion

functions are not discussed in detail because they employ no new techniques: The arcs are converted to circles, the circles are enlarged or shrunk by the `bdOuter` or `bdInner` offsets, and these offset circles are intersected to compose new arcs, three for each of the four fillets. The three arrays produced from them by the `circleseg` are concatenated, so that there are corners at the intersection points.

- lines 41-44: **Decorate the rosette**

The rosette circle is polygonized with the `circle` operator (see section 5.3.1), but the **style-rosette** function receives additionally the higher-level parameters `midpoint` and `radius` to be able to synthesize, if wanted, a more intricate window decoration. This is not possible from the polygon alone, only from the higher-level data.

- lines 48-56: **Decorate the two sub-arches**

The sub-arch is polygonized as before using `gw-polygon-2arcs-height`, but with a modified bottom height. Note that also the **style-sub-arch** function receives high-level information, one of the two sub-arches and the bottom height. They are needed to replace the standard arch by a more interesting one, using techniques like in Fig. 5.27 (2c-d)

Computation of the window geometry. At the very core of the Gothic window is the computation of the eight fields. It is performed by `gw-fillets-arcs-rosette` listed in Fig. 5.39. This function first computes a pointed arch with the same base points and excess as the main arch, but with a non-zero offset. Instead it uses the outer offset `bdOuter`. The resulting arch tightly comprises the interior fields (lines 3-4).

The base points `pLL` and `pRR` of the sub-arches are the start- and endpoints of the offset arch, moved vertically down by `arcDown` units (lines 6-7); this is the distance between the dotted horizontal lines in Fig. 5.25 (b). The other two sub-arch base points are obtained as $pRL = pM + dpM$ and $pLR = pM - dpM$ from $pM = \frac{1}{2}(pLL + pRR)$, `dpM` being the horizontal displacement vector of length $\frac{1}{2}bdInner$. Since the sub-arches have the same excess as the main arch, sufficient information is available to compute the circle segments of the sub-arcs by using again the `gw-pointed-arch` function (lines 12-16). A new operator is then employed to compute the midpoint of the rosette circle:

```
p0 p1 m0 m1 r intersect_line_ellipse → q0 q1 t0 t1
```

Computes the intersection of the line through $(p0, p1)$ with the ellipse $(m0, m1, r)$ with foci $m0$ and $m1$. It returns the intersection points $q0, q1$ and their parameters $t0, t1$ on the line $p0 + t(p1 - p0)$, so that $t0 \leq t1$. The radius r should of course be greater than $dist(m0, m1)$.

Following the analysis from Fig. 5.25 the line is the line of symmetry of the window (lines 18-19). The foci are the midpoints of the right arc of the displaced main arch, and the right arc of the left sub-arch; the radius is $rad + radL - bdInner$. Since the line of symmetry runs downwards, the first result $q0$ of `intersect_line_ellipse` is the required center of the rosette circle. The radius of the rosette is then computed as $dist(rosetteMid, m_{arcLR}) - radL - bdInner$.

A GML function can naturally have multiple return values. The end result of the function `gw-fillets-arcs-rosette` are the four arcs of the two sub-arches and the rosette circle center and radius, which are all pushed on the stack.

5.4.3 Gothic Window: Results

The efforts taken in the previous section to map the understanding of the standard Gothic window to the GML is rewarded by a very flexible and versatile modeling tool. A few example styles and profiles were created to assess the suitability of this approach.

Separation of content and appearance with window styles. Five examples for basic styles are introduced in Fig. 5.40. The most important message that the style determines only the appearance of the window. The entries in the window dictionaries (Fig. 5.30) of the windows are (row-wise) completely identical, except for the `Style` entry. As can be clearly seen, all windows realize the same excess, the same inner and outer offset distances, etc.

The same style and appearance is also preserved *throughout the recursion* (4a-e). Note that also the number of foils is identical on the different levels, if the style has a rosette with foils (4a-e, 5a-e). This is achieved not with the sub-arch-recursive function from Fig. 5.33 but with a sixth style, the *recursive* style, which uses two style entries and a window dictionary. It is set up just like in Fig. 5.31 except for two more entries:

```
/Style Style-Recursive def /TrueStyle Style-2 def /SubStyle :subwindowdict def
```

The recursive style ‘camouflages’ itself as the `TrueStyle` on the current level, and it uses the `SubStyle` window dictionary for the sub-arches – which may of course contain the recursive style again.

The purpose of the blind window style 1 in column 5.40 (a) is only to show the four types of fields for the four style functions. Note how the interior offset between the fields is realized in (3a) so that it creates a contiguous front plane. Style 2 opens the window and adds a more interesting profile. The window openings are set as rings into the back wall (3b). Style 3 uses the same profile (3c) as style 2, but it has a ‘rounded rosette’ (i.e., with round foils) in the *couronnement*,


```

1 dict dup begin !windowdict
2   /excess      1.25 def
3   /arcDown     2.0 def
4   /bdInner     0.4 def
5   /bdOuter     0.5 def
6   /wallSetback 0.1 def
7   /heightBott  2.0 def
8   /kseg        6 def
9   /Style Gothic-Window.Styles.Style-1 def
10 end
11
12 :edgeWall :edgeBack (-3,0,8) (3,0,8)
13 :windowdict
14
15 Gothic-Window.Tools begin
16   gw-gothic-window
17 end

```

Figure 5.31: Using the Gothic window modeling tool.

```

1 Gothic-Window.Styles.Style-1 begin
2
3 { usereg !edgeBack !edgeWall
4   !wallSetback !bdOuter !poly
5
6   /stdGrey setcurrentmaterial
7   :poly 5 poly2doubleface dup edgemate
8   :edgeWall killFmakeRH
9   :bdOuter 4 div :wallSetback neg 5 vector3
10  extrude
11 } /style-main-arch exch def
12
13 { usereg !edgeBack !edgeWall !poly
14   /stdGreen setcurrentmaterial
15   :poly 5 poly2doubleface !edge
16   /gold setcurrentmaterial
17   :edge edgemate :edgeWall killFmakeRH
18   [ :edge ] [ (0.05,-0.3,1) ]
19   extrudestable pop
20 } /style-fillet exch def
21
22 { usereg !rad !mid !edgeBack !edgeWall !poly
23   /stdRed setcurrentmaterial
24   :poly 5 poly2doubleface !edge
25   :edge edgemate :edgeWall killFmakeRH
26   /gold setcurrentmaterial
27   :edge (0.05,-0.3,5) extrude !edge
28 } /style-rosette exch def
29
30 { usereg !bh !arcL !edgeBack !edgeWall !poly
31   /stdBlue setcurrentmaterial
32   :poly 5 poly2doubleface !edge
33   :edge edgemate :edgeWall killFmakeRH
34   /gold setcurrentmaterial
35   :edge (0.05,-0.3,5) extrude !edge
36 } /style-sub-arch exch def
37
38 end

```

Figure 5.32: The style library for the simple style 1

```

1 Gothic-Window.Styles.Style-8 begin
2
3 { Profiles.style-main-arch-2
4 } /style-main-arch exch def
5
6 { Profiles.profile-4
7 } /style-fillet exch def
8
9 { 6 8 0.03 1.5 0.01
10 Profiles /profile-4 get
11 Profiles.rosette-pointed
12 } /style-rosette exch def
13
14 { 0.05 0.11 0.06 0.2 5
15 Gothic-Window.Styles.Style-7
16 Profiles.make-style-dict
17 Profiles.sub-arch-recursive
18 } /style-sub-arch exch def
19
20 end

```

Figure 5.33: Library of the doubly recursive style 8

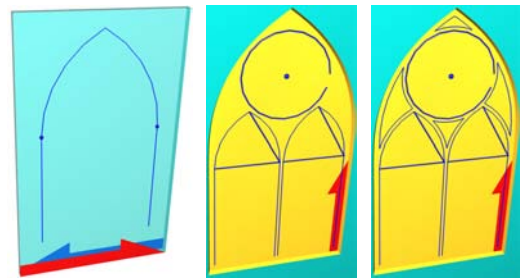


Figure 5.34: Input parameters of the style functions. The main arch polygon (a) is pushed into the wall to embed the circle and sub-arches (b) and the fillets (c)

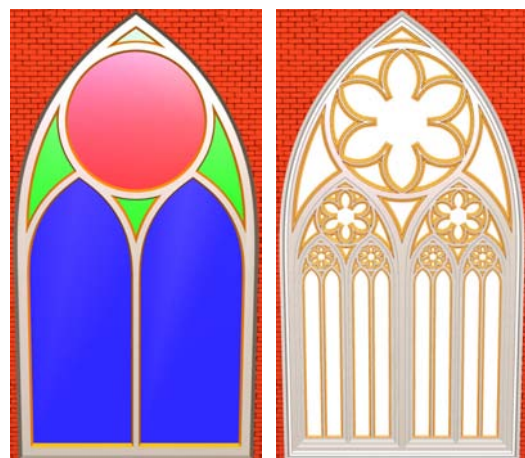


Figure 5.35: Comparison of styles 1 and 8. The simple style clearly shows the four types of fields.

```

1  usereg !windowdict
2  !pBaseR !pBaseL !edgeBack !edgeWall
3  :edgeWall facenormal !nrml
4
5  :windowdict begin Style begin
6
7  %%% DECORATE MAIN ARCH
8
9  :pBaseL :pBaseR excess 0.0 :nrml
10 gw-pointed-arch pop
11 heightBott :nrml kseg
12 gw-polygon-2arcs-height
13 bdOuter wallSetback :edgeWall :edgeBack
14 style-main-arch !edgeArch
15
16 %%% COMPUTE SUB-ARCS AND ROSETTE
17
18 :nrml wallSetback neg mul dup
19 :pBaseL add !pL
20 :pBaseR add !pR
21
22 :pL :pR :nrml
23 gw-compute-arcs-rosette
24 !rosetteRad !rosetteMid
25 !arcRR !arcRL !arcLR !arcLL
26
27 %%% COMPUTE AND DECORATE THE FOUR FILLETS
28
29 :arcLL :arcLR :arcRL :arcRR
30 :rosetteMid :rosetteRad :pL :pR :nrml
31 gw-compute-fillets
32 gw-polygon-fillets 4 array
33 { :edgeArch :edgeBack style-fillet }
34 forall
35
36 %%% DECORATE THE ROSETTE
37
38 :rosetteMid :nrml :rosetteRad kseg 4 mul
39 circle
40 :edgeArch :edgeBack :rosetteMid :rosetteRad
41 style-rosette
42
43 %%% DECORATE THE TWO SUB-ARCHES
44
45 heightBott bdOuter add !heightInnerArc
46
47 :arcRL :arcRR :heightInnerArc :nrml kseg
48 gw-polygon-2arcs-height
49 :edgeArch :edgeBack :arcRL
50 style-sub-arch
51
52 :arcLL :arcLR :heightInnerArc :nrml kseg
53 gw-polygon-2arcs-height
54 :edgeArch :edgeBack :arcLL
55 style-sub-arch
56
57 end end

```

Figure 5.36: Main window function gw-gothic-window

```

1  usereg !nrml !offset !excess !pR !pL
2
3  :pR :pL :excess line_2pt !mR
4  :pL :pR :excess line_2pt !mL
5  :pL :pR dist :excess mul :offset sub !rad
6
7  :mL :rad :mR :rad :nrml
8  intersect_circles pop !qT
9
10 [ :qT :mL :pL :pR :offset move_2pt ]
11 [ :pR :pL :offset move_2pt :mR :qT ] :rad

```

Figure 5.37: Function gw-pointed-arch

```

1  usereg !kseg !nrml !hb !arcR !arcL
2
3  [ :arcR 0 get :hb putZ dup ]
4  :arcR :nrml :kseg 1 circleseg arrayappend
5  :arcL :nrml :kseg 1 circleseg arrayappend
6  [ :arcL 2 get :hb putZ dup ] arrayappend

```

Figure 5.38: Function gw-polygon-2arcs-height

```

1  usereg !nrml !pR !pL
2
3  :pL :pR excess bdOuter :nrml
4  gw-pointed-arch !rad !arcR !arcL
5
6  :arcL 2 get (0,0,-1) arcDown mul add !pLL
7  :arcR 0 get (0,0,-1) arcDown mul add !pRR
8  :pLL :pRR midpoint_2pt !pM
9  :pRR :pLL sub
10 bdInner 0.5 mul setlength_vec !dpM
11
12 :pLL :pM :dpM sub excess 0.0 :nrml
13 gw-pointed-arch !radL !arcLR !arcLL
14
15 :pM :dpM add :pRR excess 0.0 :nrml
16 gw-pointed-arch !radR !arcRR !arcRL
17
18 :pM (0,0,1) add
19 :pM (0,0,1) sub
20 :arcLR 1 get
21 :arcR 1 get
22 :rad :radL add bdInner add
23 intersect_line_ellipse 3 pops !rosetteMid
24
25 :rosetteMid :arcLR 1 get dist
26 :radL sub bdInner sub !rosetteRad
27
28 :arcLL :arcLR
29 :arcRL :arcRR
30 :rosetteMid :rosetteRad

```

Figure 5.39: Function gw-compute-arcs-rosette

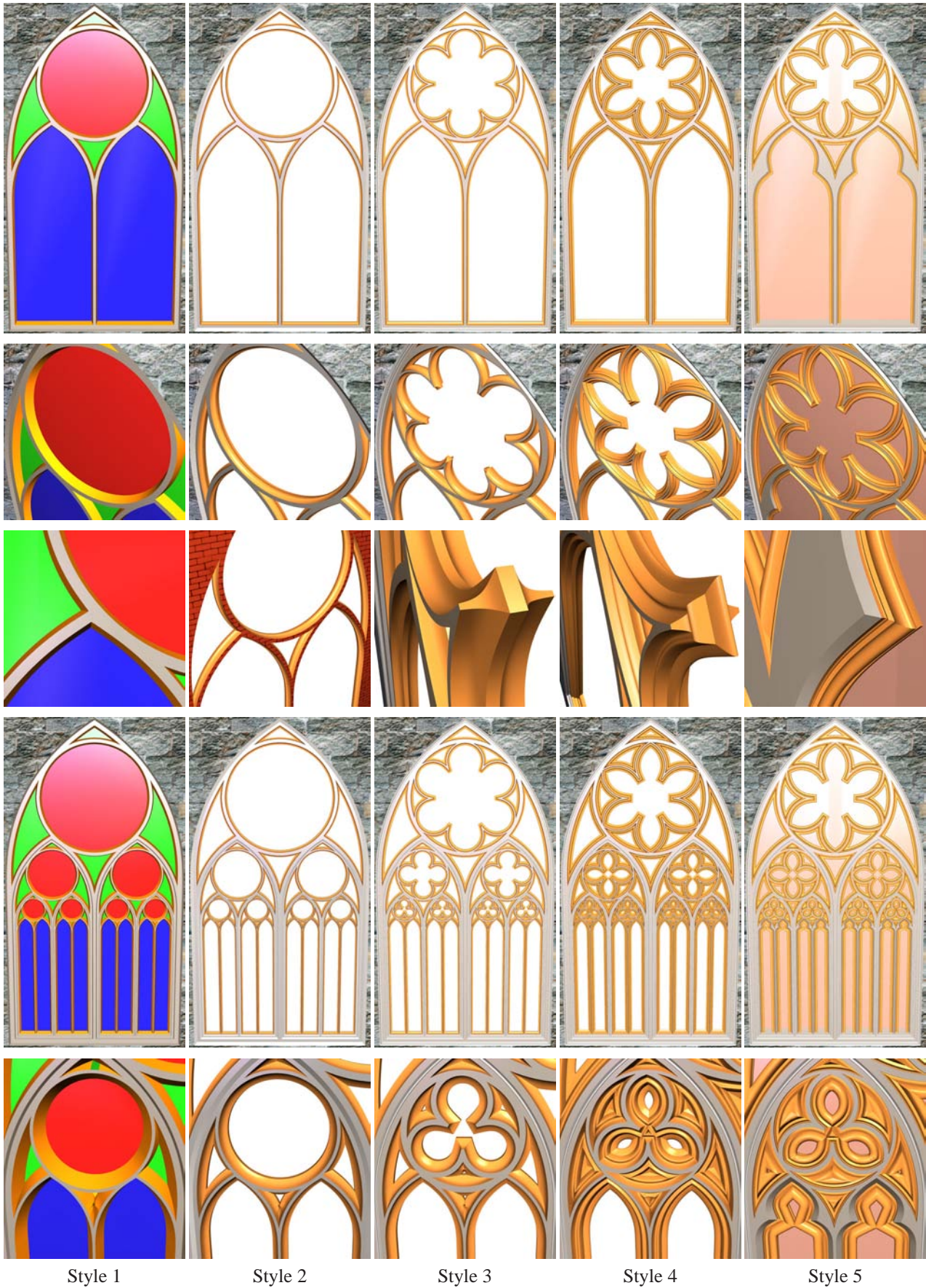


Figure 5.40: Five basic Gothic window styles.

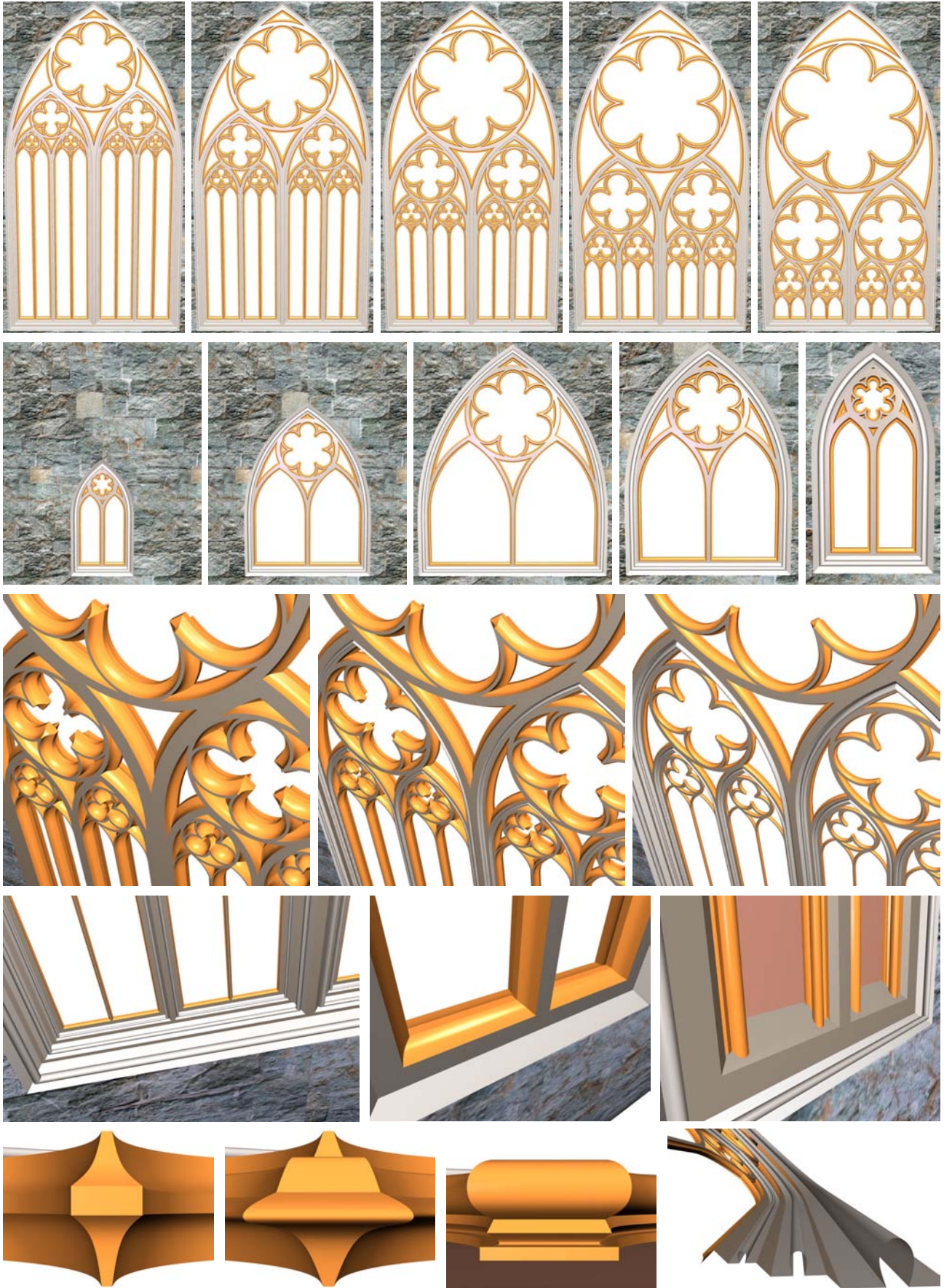


Figure 5.41: Parameter variations in a manifold of Gothic windows.

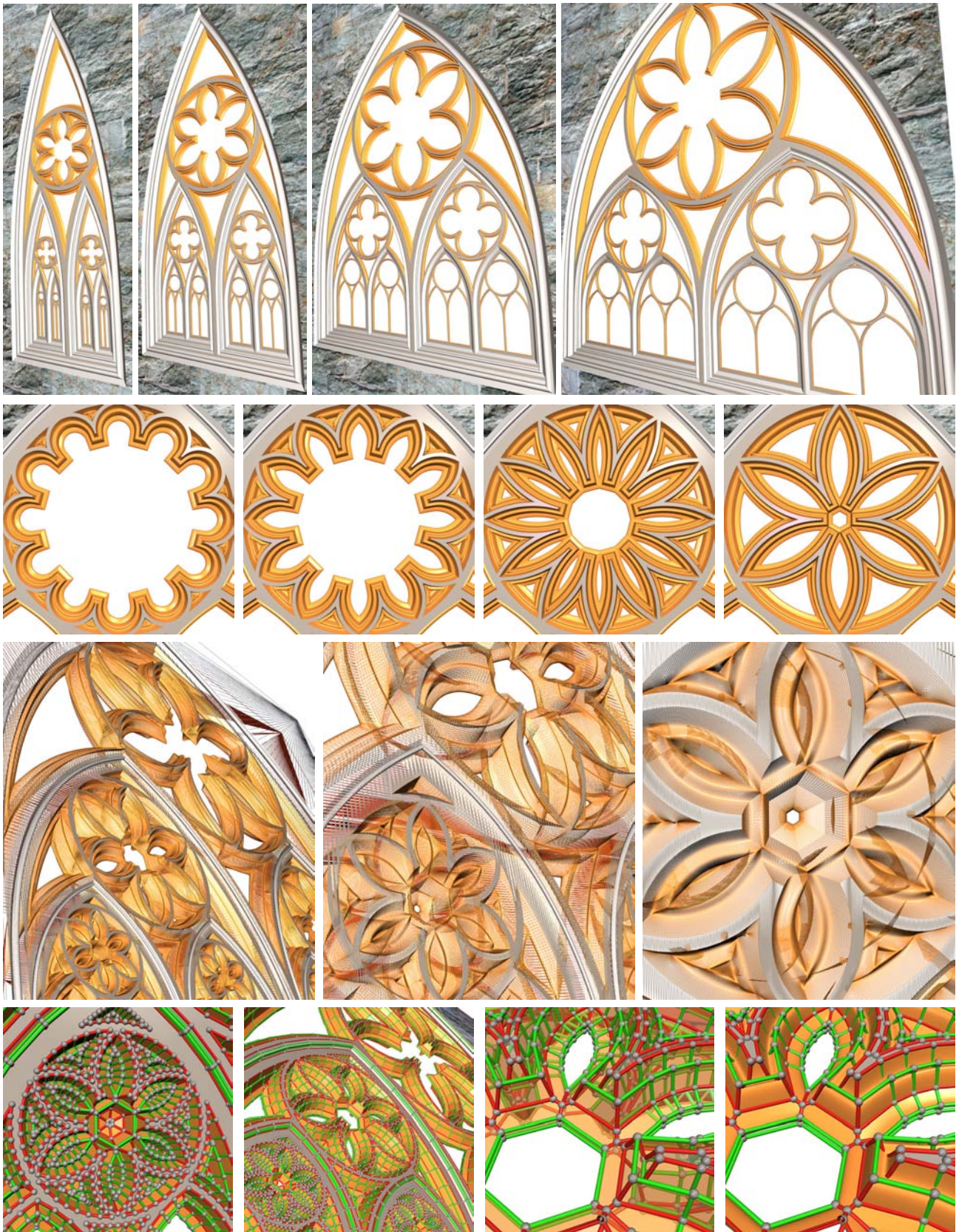


Figure 5.42: The principle of information unfolding applied: High output complexity from few input parameters such as width, height, and base height.

and the main arch profile is more complicated (also shown in Fig. 5.41 (5d)). Style 4 introduces the pointed rosette, and it uses a much more interesting profile (3d). This profile ranges much farther than the rosette profile from style 2, which implies that it must be applied using stable extrusion instead of the standard extrude operator.

`[e0 .. ek][p0 .. pl] extrudestable → [e0 .. ek]`

realizes the intersection free extrusion based on the straight skeleton from section 4.5.5. The input faces e_i must be individual faces or rings of the same plane; they may not be direct neighbours or (pairwise) share a vertex. The profile points $p_i = (x_i, y_i, z_i)$ specify the horizontal and vertical offset as explained in 4.5.5.

The benefit of using the intersection-free extrusion can be nicely seen in Fig. 5.40 (5d) and (5e): The style 5 uses also the pointed rosette, but with again another profile. It also introduces a new feature, since it replaces the standard arch with the pointed trefoil arch from Fig. 5.27 for the sub-arches (3e). This is an example of utilizing the high-level information passed on to style-sub-arch, a fact that was already mentioned several times. In style 5 this style function replaces the already sampled pointed sub-arch.

Parameter variations: A manifold of windows, and the window manifold. Six of the eight parameters in a window dictionary have a geometric meaning; and even more geometric parameters can be added by the style, for instance the number of foils when the style employs a rosette. In conjunction with recursion, this permits to set up interesting dependencies between the variables on the different levels. Some possibilities are illustrated in Fig. 5.41.

Row 1 of Fig. 5.41 shows the effect of an increasing vertical offset of the sub-arch base points with respect to the base points of the main arch in a doubly recursive window. The size of the windows in the sub-arches is only half the size of the higher level. For the whole ensemble to be harmonic the sub-windows should therefore have half the vertical offset of the main window. This rule binds the vertical offset of the sub-windows on both levels of recursion, which were free parameters before. The only remaining free high-level parameter is the vertical offset of the main window.

Fig. 5.41 (1e) shows a geometric degeneracy: The right and left fillets are only made of three circle segments but for larger vertical base point offsets they should also contain a straight line segment. The problem is that the fillets are explicitly constructed by circle offsets in the way explained in 5.4.2. Also other degeneracies are possible, for instance the top fillet can completely vanish for four-centered arches (Fig. 5.23 (1a)) with a small excess < 1 . But in reality the fillets are only the remains of the front plane when the circle and the sub-arches are removed. So the solution would be to use the appropriate method, i.e., a 2D-CSG algorithm that can handle curves made not only of line segments but also of circle segments. This is planned in the future.

Row 2 of Fig. 5.41 shows the effect of *proportions*. The three walls in 5.41 (a), (b), and (c) have all the same absolute size; so (a) shows a small window, (b) a window of medium size, and (c) a large one. The windows in (d) and (e) are scaled: (e) is the window from (a) scaled to the size of (c), and (d) is the window from (b) scaled to the size of (c). In absolute units the width of the bars in all windows is the same (same `bdInner`). This leads to the (correct) impression that 5.41 (e) looks like a 'baby window' compared to (c) only because of its different proportions.

Row 3 of Fig. 5.41 shows the effect when the main arch of a doubly recursive window is increasingly pushed into the wall. Following the same approach as before the `wallSetback` of the sub-windows is (recursively) half the `wallSetback` of the window from one level higher. So only the `wallSetback` of the highest level is a free parameter. The result of this rule is shown in (5d), the size of the main arch profile decreases by a factor of two with each level of recursion. As the `wallSetback` increases from (3a) over (3b) to (3c) it also has an effect on the profiles: They become thinner, and the profiles of the sub-arches become proportionally thinner as well. The reason is that the style functions from style 3 (in fact from all styles) measure the distance between the front- and backside planes. This determines a scale factor for the profiles, so that the x/y proportions of the profile remain the same, i.e., they always look as in 5.41 (5a-c).

Row 3 of Fig. 5.41 shows the windowsills of the different styles. Styles 1-4 use face extrusion, i.e., either the `extrude` or `extrudestable` operator, to apply the profile. Style 5 is different, since it uses *path extrusion* from section 4.5.4. The benefit is that this style can provide a beautiful windowsill (4c). The drawback is that path extrusion can not yet handle self intersections as in 5.40 (5e). Path extrusion with removal of self intersections is future work.

Information unfolding: High output complexity from few input parameters. The benefit of a generative shape description is most concisely expressed with the unsqueezing of the doubly recursive Gothic window in Fig. 5.42 (1a-c) with three different window styles on its three levels. The pointed arch, the technological breakthrough from the 12th century, keeps its promise that the arch base points remain on the same height when the width of the window varies. The rosette on the next row Fig. 5.42 (2a-d) shows the transition from round foils to increasingly pointed foils. Only two parameters are varied, the pointedness and the number of foils.

Finally the effectiveness of the *principle of information unfolding* is demonstrated with the doubly recursive style 4 window in row 3 of Fig. 5.42. The density of the wire frame indicates the overwhelming amount of data produced: Image (3c) shows only the smallest of the three levels, but in highest refinement, with each smooth face four times subdivided.

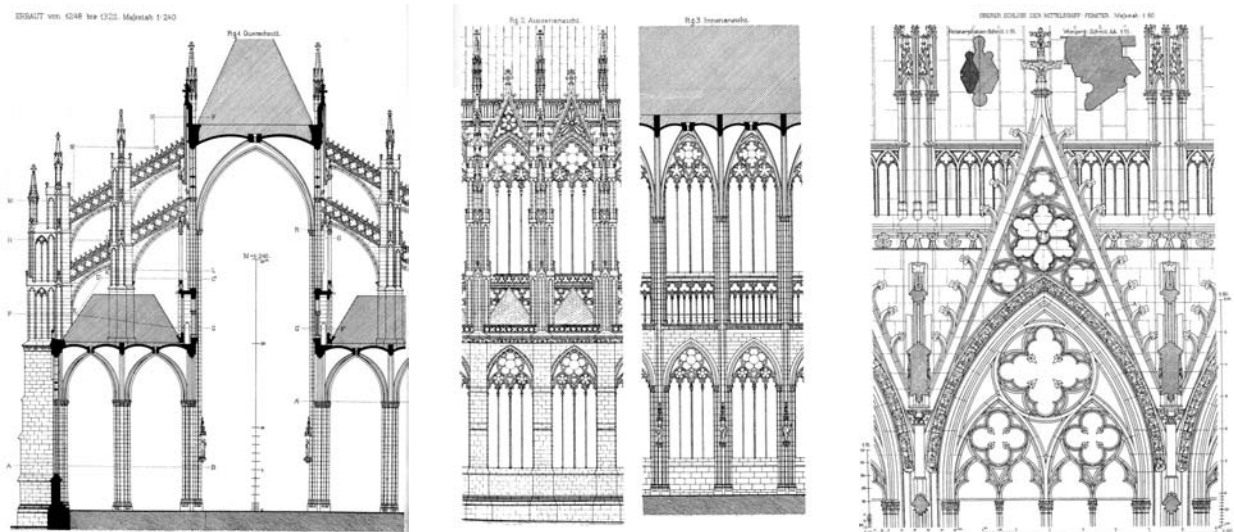


Figure 5.43: Plans of the cathedral in Cologne (Germany). Sections through the nave (a), the walls of the nave from outside (b) and inside (c), and a typical decoration of Gothic windows: A *canopy*, also called *Gothic gable*. Illustrations are from Egle [EF05].

5.4.4 The Procedural Cathedral

The cathedral of Cologne is one of the greatest Gothic cathedrals in the world. Impressive is its incredible size, with a height of 157 meters, a length of 145 meters and a width of 86 meters. But equally impressive is the overwhelming wealth of architectural detail that apparently dissolves its huge mass of 160.000 metric tons of stone. This lightness and fragility, that even seems to overcome gravity, was a primary goal of its founding builders. The cathedral was begun in 1248 when Gothic architecture had its high time across all Europe. But the project was so gigantic that in the first 150 years of construction only the foundations of the two towers and the choir were erected, and in 1410, due to the fading faith at this time, the project came to a standstill – for 432 years. In the early 19th century Geheimrat Johann W. v. Goethe, the famous poet, brought his influence as Prussian government official to bear, and in a period of German national enthusiasm the cathedral was eventually finished in 1841-1880. In this last period the builders faithfully followed the original medieval plans, but they could also utilize new efficient methods from the emerging industrial age. As a result the Cologne cathedral exhibits unmatched stylistic purity and clear symmetry – unlike other medieval cathedrals that were built continuously for so many years that all changes in taste and fashion left their traces on them.

Structure on every level. The cathedral incessantly invites the spectator to decipher its structure, to speculate whether a certain element might be supporting or just decoration, which measures are determined through constructional reasons, and which parts are necessary because of style and symmetry. What becomes very clear is that nothing is really redundant, and no part is just arbitrarily added. The whole cathedral is formed by an intricate composition that determines which ornaments are to be placed in which positions, which pointed arches are crowned by Gothic gables and which are not, how to calculate the radii of the circles, and which parts to decompose further into sub-parts. Yet still it is quite enjoyable to look for ‘errors’ in the building, since there are quite a few (see Fig. 5.44 (2d)).

It is most fascinating how new structures appear on every refinement level: From a distance only the two towers and the cross-shaped ground layout with the nave and the two transepts are perceived. Then it becomes obvious that the cathedral has two aisles on each side of the nave. The nave with its height of 43,35 meters is supported by two rows of flying buttresses that emerge from the aisles. Immense canopies span over enormous windows in the shape of pointed arches, filled by rich and detailed window tracery. Coming closer the tracery exhibits even more detail: The whole cathedral is covered over and over by details on a centimeter scale, for instance around 11.000 little towers, each only ≈ 15 cm high.

There is no precise 3D model yet. This building is a very good example for a shape where manual CAD modeling as well as 3D scanning, as explained earlier in 10, are infeasible. The fotos in Fig. 5.44 give an idea why still as of today, no precise 3D model of the Cologne cathedral exists. To obtain a good end result both approaches would require very much manual refurbishing and post-processing. This makes them so cost intensive that for obtaining only a single model, even if it is the Cologne cathedral, this effort is also hard to justify.

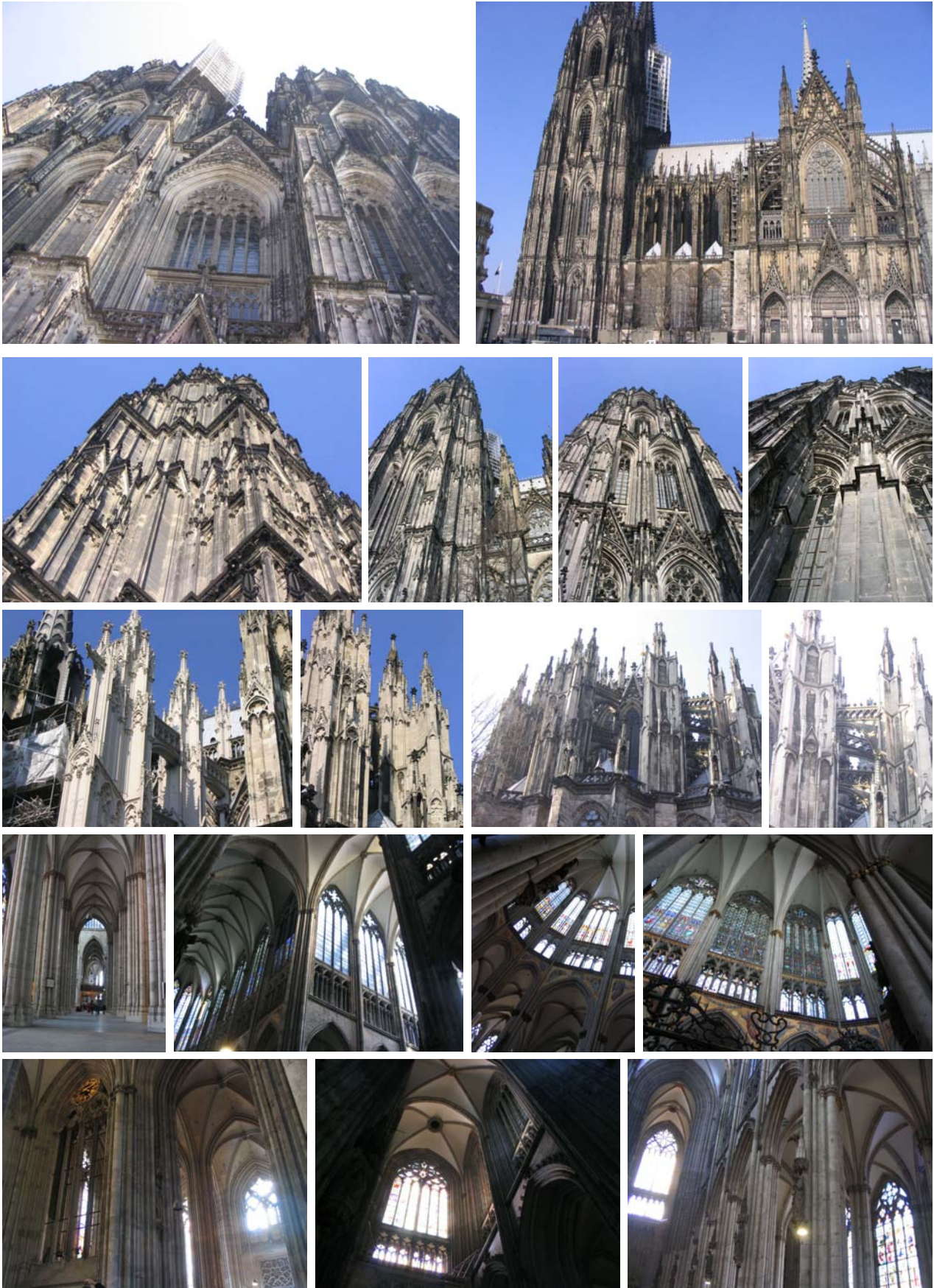


Figure 5.44: Fotos from the Cologne cathedral. Truly remarkable is the impact of the light that enters through the huge windows (lower rows).

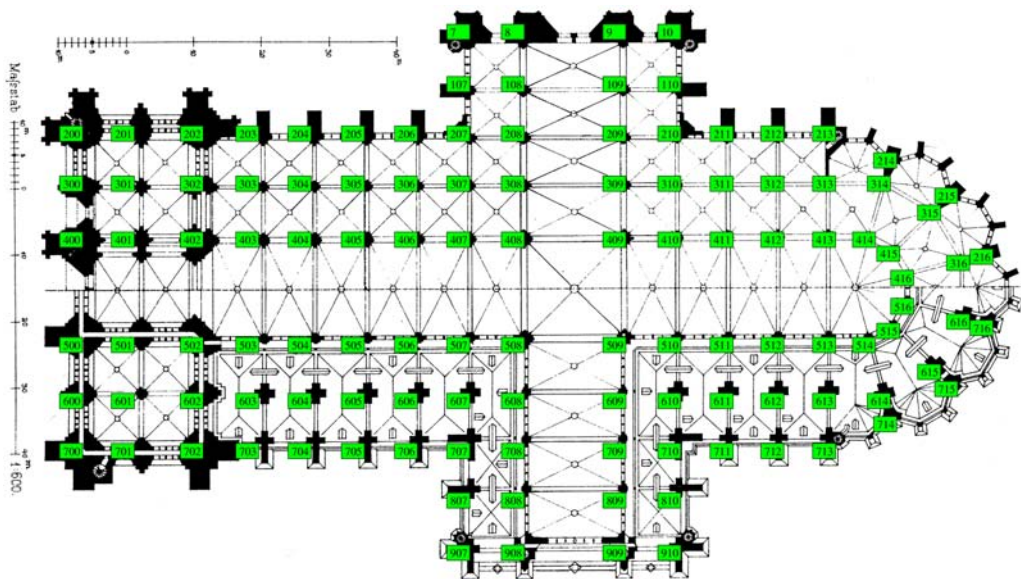
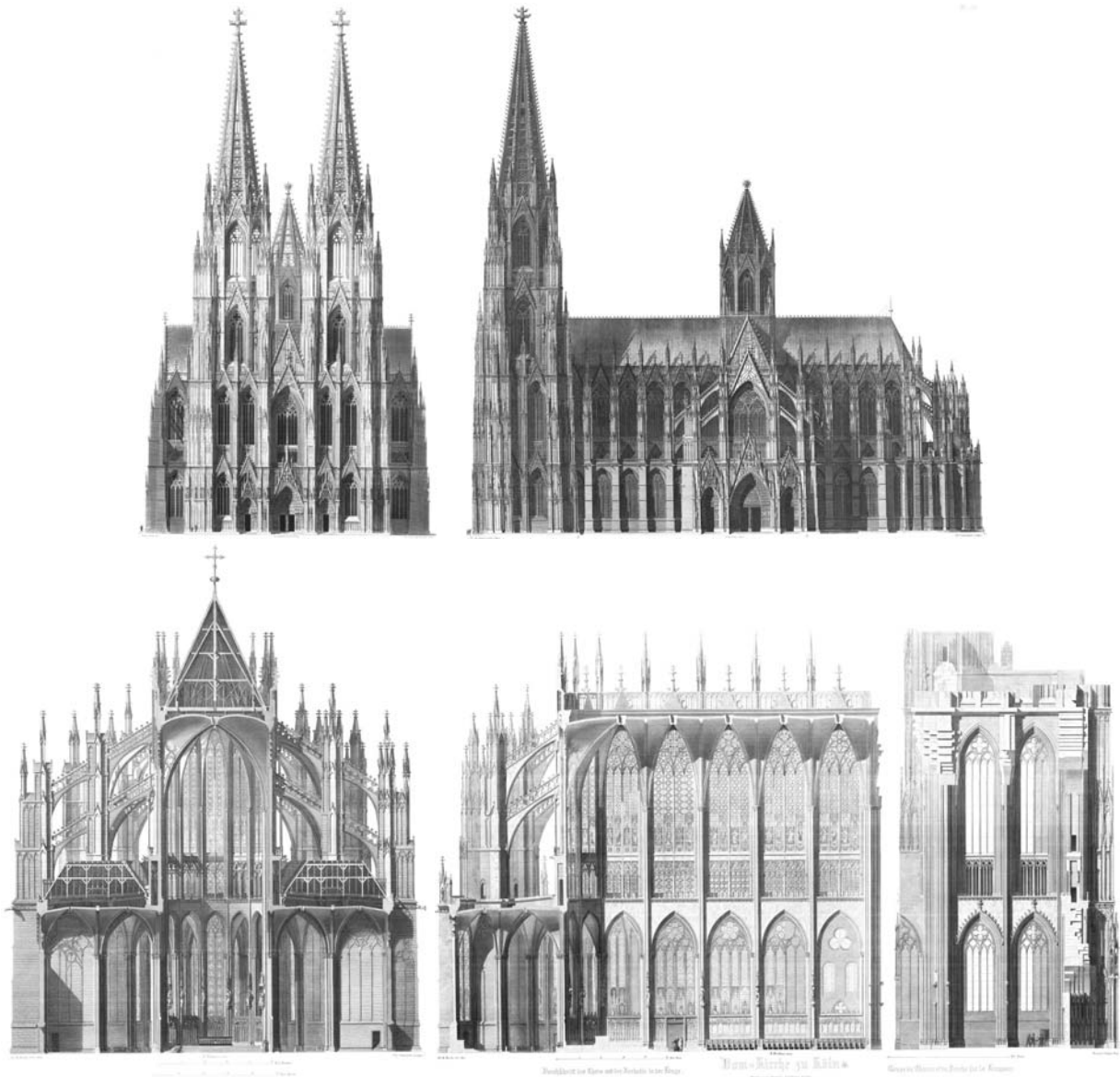


Figure 5.45: Plans of the Cologne cathedral. Row 3: Numbering scheme for the grid of pillars.

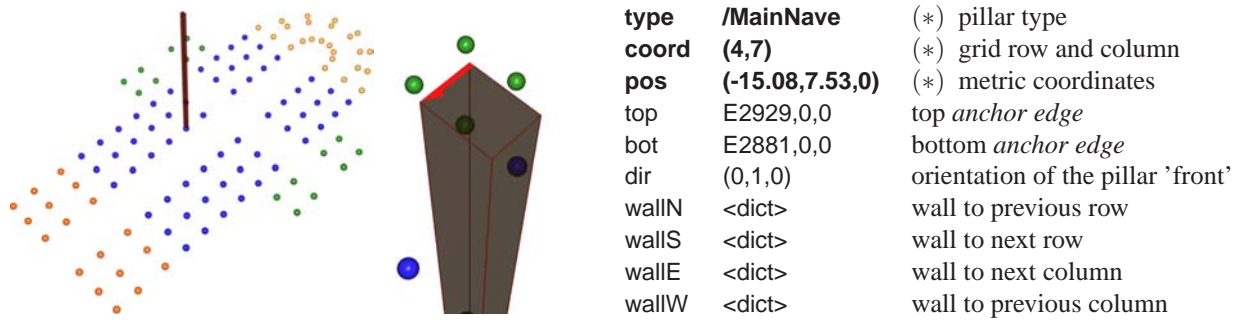


Figure 5.46: Entries of a pillar dictionary. The example pillar 407 is in row 4, column 7 (zero-based counting).

Generative reconstruction of the Cologne cathedral. A faithfully produced procedural cathedral model would be most beneficial, not just to obtain a digital model that is electronically accessible and immediately intelligible in 3D. A much stronger incentive is the scientific challenge: The hypothesis that Gothic buildings are based on a limited set of construction principles can be proven only if it is possible to formulate these rules also with a concrete building at hand. The measure for success is the information reduction that is possible with a generative description of the Cologne cathedral: How many tokens are necessary to describe it? – Hopefully much fewer than the number of triangles, because it is probably gigantic.

But if this endeavour has a positive result then not only the model of a single cathedral is obtained by it. With a first generative cathedral at hand the reconstruction of the second, the third, and the following cathedrals and churches becomes ever more manageable and efficient. So the benefit would be both practical and theoretical: Obtaining re-usable tools, and a better understanding of Gothic architecture. – But like with the pointed arch and the Gothic window, the analysis must precede the synthesis.

The grid of pillars. Fotos such as those in Fig. 5.44 give a great impression and a feeling how the church looks like. They can also be used to resolve detail problems with the construction. But it is very hard to derive from them exact measurements, and in particular the high-level parameters for generative models; in fact this is part of future work. Better suited as a source for the reconstruction are plans. Some of the best illustrations were in fact made for marketing reasons. Sulpiz Boisseree, a merchant from Cologne, has invested much time and money to let create twenty impressive engravings that convinced his contemporaries that the cathedral should be finished [Boint] – for example J.W. v. Goethe. Some of the engravings are shown in Fig. 5.45; some details differ from the realized building, e.g., the crossing tower.

Row 3 shows the ground layout of the cathedral, it is again taken from Egle [EF05]. As explained in detail by Prof. Thies the principal structuring element in a cathedral are the pillars. For the first reconstruction they were numbered in the form of a grid, shown with the green markers, in the following way.

- Columns are counted from N to S, rows from W to E, counting starts at zero
- Pillar IDs such as 407 refer to the pillar in row 4, column 7
- Rows continue also in the choir so that three pairs of pillars (416,516), (316,616), (216,716) are connected
- Rows 0, 1 have entries only in the northern transept with pillars 7, 8, 9, 10 and 107, 108, 109, 110
- Rows 8, 9 have entries only in the southern transept with pillars 807, 808, 809, 810 and 907, 908, 909, 910
- The pillars in the corners of central square, the *crossing piers*, have IDs 408, 409, 508, 509 (NW, NE, SW, SE)

The generative model. The pillar grid is realized in the GML as a 2-dimensional array of dictionaries with one dictionary per pillar. An example is shown in Fig. 5.46. Mandatory entries are bold: type, coord, and pos, the pillar positions shown as dots in 5.46 (a). The other entries are added when geometry is created for the pillar. The non-existing entries in the grid ((0,0)-(0,6) etc.) all refer to the same dummy pillar dictionary.

The pillar type is set in an initializing function `set-type-pillars`, part of which is shown in Fig. 5.50. Schematic information can be set using a loop, in this case over the grid columns since, e.g., most of the columns in row 4 are of type `Pillar-Nave` (line 4). The `getcol` function is extremely useful as it allows to directly access each pillar; it expects row and column and returns the pillar dictionary. Its implementation is almost trivial (Fig. 5.54) as it uses `Pillars`, the array of arrays of pillar dictionaries. The orientation `dir` of the pillars is important because they are usually not symmetric as they will be attached to different types of walls. By convention the pillars are directed *outwards*, i.e., the northern pillars are directed northwards to (0, 1, 0). The eight different pillar types are listed in Fig. 5.56. Note that the transepts and the crossing require different pillar types in rows 3 and 4; but the entries in the pillar dictionaries can simply be overwritten.

```

usereg
0 1 16 { !i
  0 1 9 {
    :i getcol
    dup .type
    load exec
  } for
} for

```

Figure 5.47: create-pillars

```

usereg
0 1 16 { !i
  0 1 9 { :i getcol !col
    :col /wallN build-wall
    :col /wallS build-wall
    :col /wallW build-wall
    :col /wallO build-wall
  } for
} for

```

Figure 5.48: create-walls

```

usereg !wall !col
:col :wall known {
:col :wall get !walldict
:walldict /front known not {
:walldict dup /type get
load exec
} if
} if

```

Figure 5.49: build-wall

The wall dictionaries, and navigating in the grid. One dictionary is not only made for each pillar but also for each wall between the pillars. The wall and pillar dictionaries contain direct references to each other. The set-type-walls function from Fig. 5.51 is similar to the set-type-pillars function from before in that it also loops over the columns to set the wall type (sometimes preliminarily). A pillar can be adjacent to four walls at most, to the absolute directions north, east, south, and west within the grid. This corresponds to the wallN, wallE, wallS, wallW entries in the pillar dictionaries. If a pillar has no wall in some direction, like the northern buttress (2,7) that has no northern wall, the respective dictionary entry is simply missing. – As a remark, in OOP (C++) all objects that belong to a class have the same member variables and functions; with GML dictionaries used as class objects this does not have to be the case.

Each wall is attached to two pillars, to its right and left, but for walls right and left are only relative. A wall is set up with the make-wall function from Fig. 5.52 which expects two pillar index pairs of the *left* and *right* pillars and a wall type, in this order. There are two sorts of walls, Wall-EW-*x* in east-west direction and Wall-NS-*y* in north-south direction, and several types of each sort. As an example, the wall going east from the NE crossing pier (4,9) has this pillar as its left (colL) and (4,10) as its right (colR). The wall going north from the same crossing pier (4,9) goes to pillar (3,9), which is the colL of the wall in between. The wall orientation with right/left is defined in set-type-walls as follows:

- for Wall-NS-*y* the *northern pillar is left* (lines 2-5)
- for Wall-EW-*x* in the northern church half the *western pillar is left* (lines 7-9)
- for Wall-EW-*x* in the southern church half the *eastern pillar is left* (lines 10-12)

In summary it is effectively possible to navigate in the grid by going from one pillar to the next over the walls:

```
4 9 getcol .wallE .colR .wallN .colL .wallW .colL is the yields the same result as 3 9 getcol
```

This means that to follow the walls east, north, west is just as good as to go north directly.

The semantic network. So far only the network of dictionaries has been set up, not the slightest bit of geometry is created. This strategy has proven extremely valuable for more complex constructions because the description can still be changed on a high level. Data can be added to the wall and column dictionaries to provide sub-sub-styles with supplemental information. As an example many of the walls have Gothic windows; and of course it is desirable to re-use window styles from the previous section 5.4.2. The set-type-walls function shows how to achieve this: The return value of make-wall is the newly created wall dictionary. Usually it is just popped, but in lines 8, 10, 11, and 13 of set-type-walls it is used to enter a particular window style dictionary (as set up in 5.31) for the wall.

The semantic network permits to change specific data also after everything has been set up and initialized, but before the geometry is created. As an example, it is still possible to set the style of just one particular window to a different style. – Another great advantage is that the network also permits efficient access to the geometry when it is created.

A second or two to build a cathedral. When the semantic network is properly set up two very simple functions are executed to actually create the mesh, first create-pillars from 5.47, then create-walls from 5.48, which in turn uses build-wall from 5.49. These functions reveal that the pillar and wall types are more than just names: They are functions that are loaded and executed. Their respective arguments are the pillar/wall dictionaries they were taken from. To create two pillars and a single wall between them is as simple as, e.g.,

```
4 7 getcol dup .type load exec 4 6 getcol dup .type load exec 4 7 getcol /wallW build-wall
```

One example of a simple pillar generating function is Pillar-Nave listed in Fig. 5.55. It creates a box shape by triple extrusion, removes some of the edges at the side of the box, and stores the top and bottom faces in the pillar dictionary. They serve as *anchor edges* for the next construction steps; it is therefore important that the anchor edges are well specified and reliably noted. – An important result of this first crude reconstruction is that the required set of anchor edges must be more carefully selected and specified. Felix Funke and Florian Rudolph are currently taking care of this issue.

```

3 1 13 { !i
  2 :i getcol dup /type /Pillar-Buttress put /dir (0,1,0) put
  3 :i getcol dup /type /Pillar-Aisle put /dir (0,1,0) put
  4 :i getcol dup /type /Pillar-Nave put /dir (0,1,0) put
  5 :i getcol dup /type /Pillar-Nave put /dir (0,-1,0) put
  6 :i getcol dup /type /Pillar-Aisle put /dir (0,-1,0) put
  7 :i getcol dup /type /Pillar-Buttress put /dir (0,-1,0) put
} for

```

Figure 5.50: Part of the function set-type-pillars

```

3 1 12 { dup !i0 1 add !i1
  2 :i0 3 :i0 /Wall-NS-Buttress make-wall pop
  3 :i0 4 :i0 /Wall-NS-Aisle make-wall pop
  4 :i0 5 :i0 /Wall-NS-Nave make-wall pop
  6 :i0 5 :i0 /Wall-NS-Aisle make-wall pop
  7 :i0 6 :i0 /Wall-NS-Buttress make-wall pop

  2 :i0 2 :i1 /Wall-EW-Aisle-Window make-wall /Style :windowstyle put
  3 :i0 3 :i1 /Wall-EW-Aisle-Arch make-wall pop
  4 :i0 4 :i1 /Wall-EW-Nave-Window-Canopy make-wall /Style :windowstyle put
  5 :i1 5 :i0 /Wall-EW-Nave-Window-Canopy make-wall /Style :windowstyle put
  6 :i1 6 :i0 /Wall-EW-Aisle-Arch make-wall pop
  7 :i1 7 :i0 /Wall-EW-Aisle-Window make-wall /Style :windowstyle put
} for

```

Figure 5.51: Part of the function set-type-walls

```

1 usereg
2 !walltype !bx !by !ax !ay
3
4 :ax :ax getcol !colA
5 :by :bx getcol !colB
6
7 dict dup begin !wall
8 :colA /colL edef
9 :colB /colR edef
10 :walltype /type edef
11 end
12
13 :ax :bx sub !dx
14 :ay :by sub !dy
15
16 :dx -1 eq /wallW if
17 :dx 1 eq /wallO if
18 :dy -1 eq /wallN if
19 :dy 1 eq /wallS if !wallB
20
21 :dx -1 eq /wallO if
22 :dx 1 eq /wallW if
23 :dy -1 eq /wallS if
24 :dy 1 eq /wallN if !wallA
25
26 :colA :wallA :wall put
27 :colB :wallB :wall put
28
29 :wall

```

Figure 5.52: make-wall is called by set-type-walls in Fig. 5.51 above

```

dict dup begin
  /pos (0,0,0) def
  /coord (0,0) def
  /type /pop def
end

```

Figure 5.53: make-pillar

```

exch Pillars
exch get
exch get

```

Figure 5.54: getcol

```

1 usereg !col
2
3 :col .pos !p
4 :col .dir dup Hpfeiler-breit 0.5 mul mul !w
5 (0,0,-1) cross Hpfeiler-schmal 0.5 mul mul !h
6
7 [ :p :w :h add add
8 :p :w neg :h add add
9 :p :w neg :h neg add add
10 :p :w :h neg add add ]
11 3 poly2doubleface
12
13 dup edgemate faceCCW
14 :col /bot exch put
15 [ (0,20,3) (0,26,3) (0,46,3) ] extrude faceCW
16 :col /top exch put
17
18 :e faceCCW dup kill-side-edges
19 faceCCW dup kill-side-edges
20 faceCCW kill-side-edges

```

Figure 5.55: Pillar-Nave as a pillar example; set as pillar type of rows 4,5 in set-type-pillars in Fig. 5.50 above

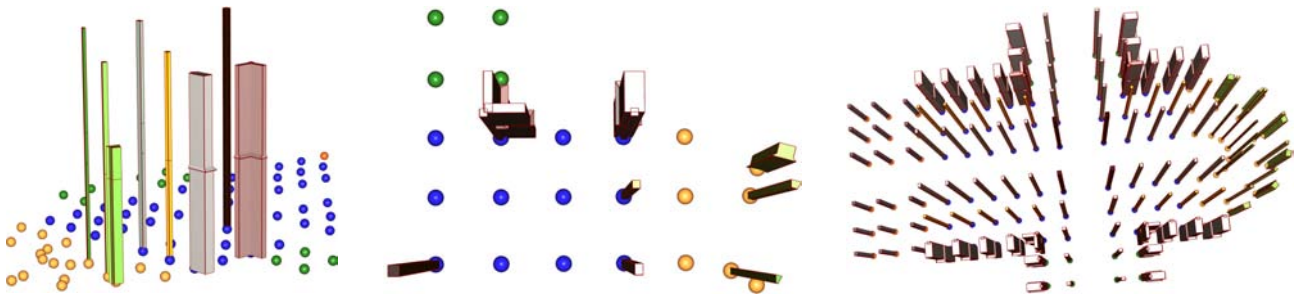


Figure 5.56: The eight different pillar types. Twice the basic types on rows 2,3,4, once in the nave and once in the choir, plus the crossing pier and the two-direction buttress.

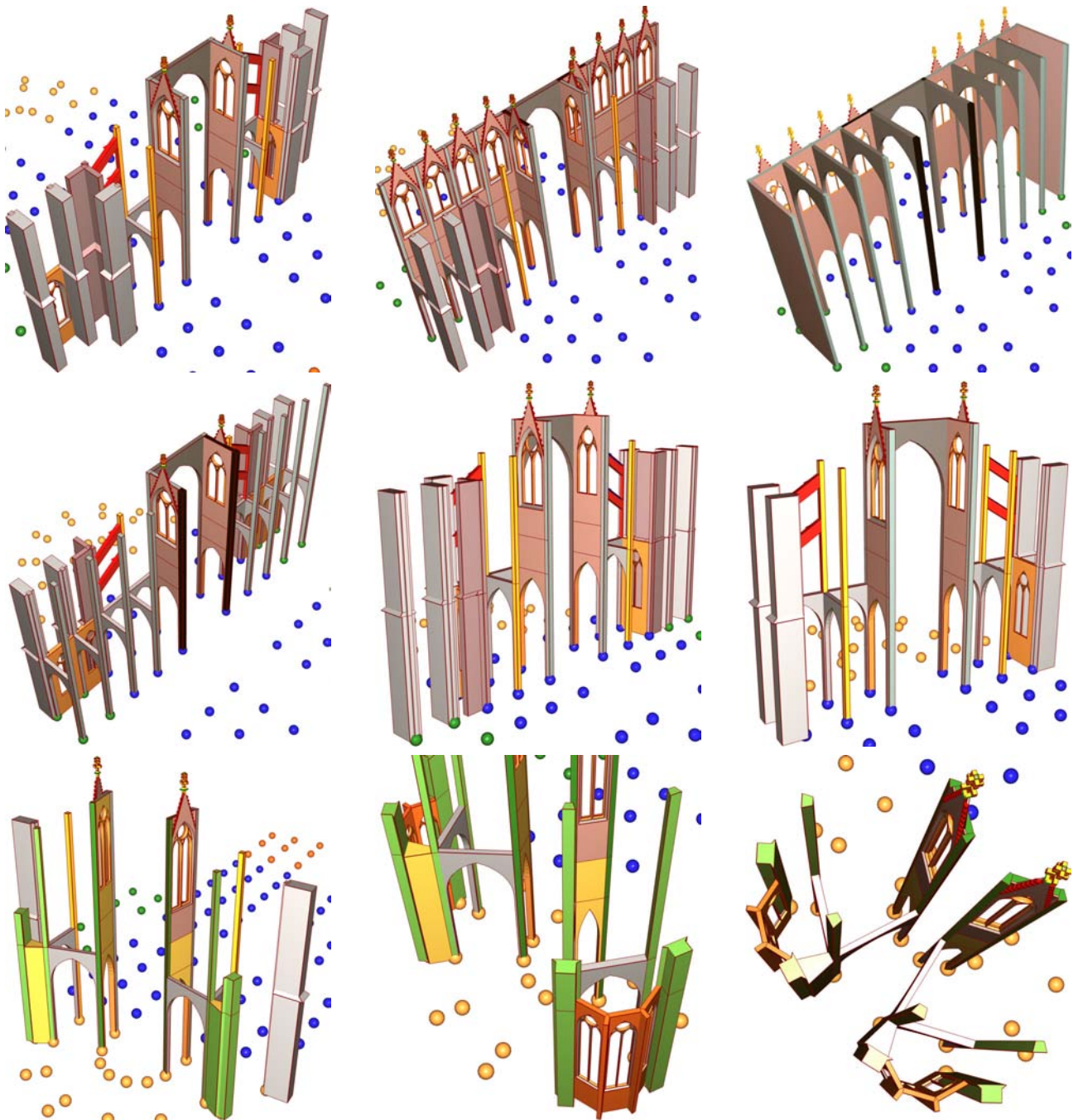


Figure 5.57: The different wall types.

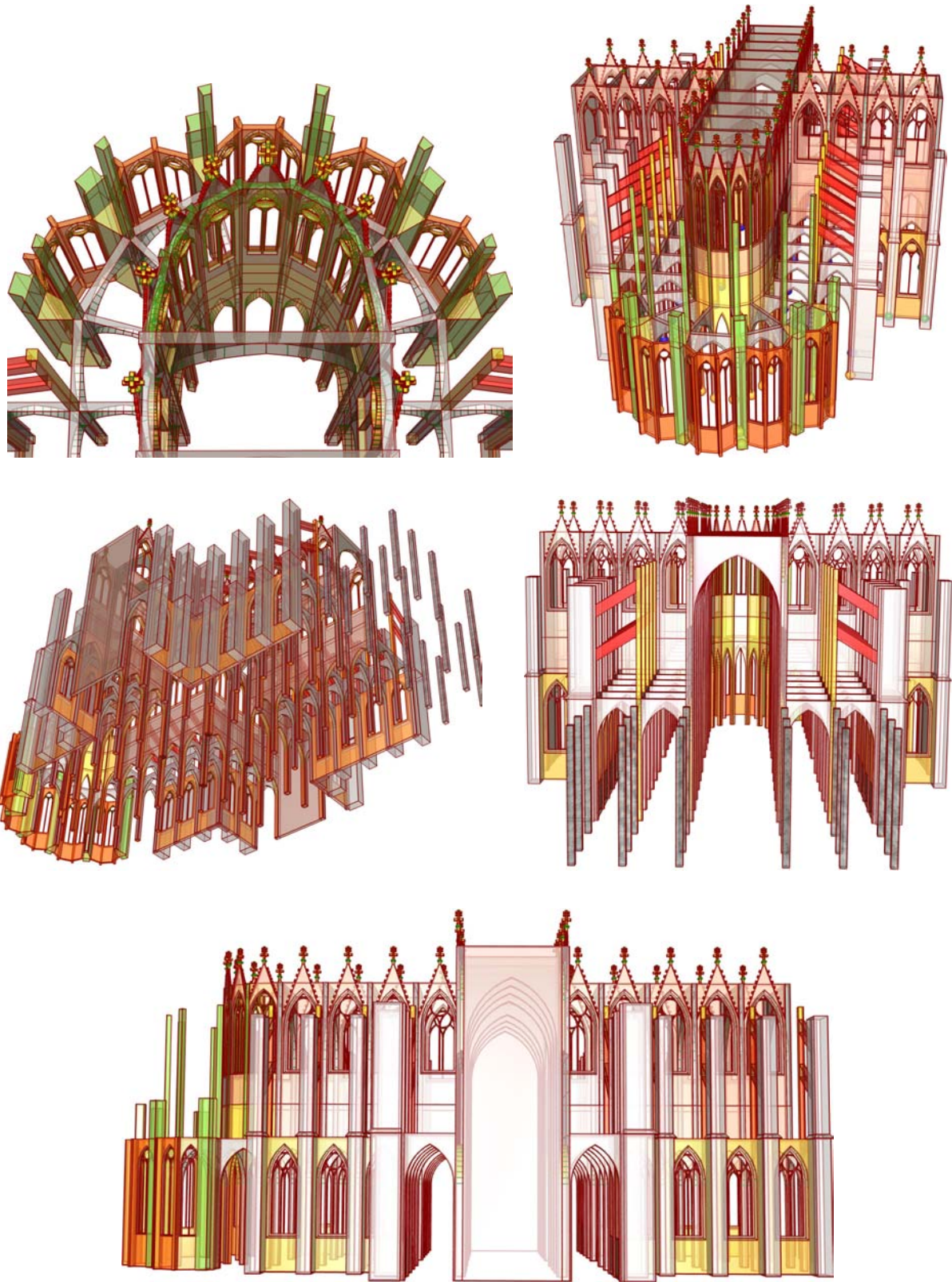


Figure 5.58: The transparent cathedral. The Gothic gables above the windows were provided by Florian Rudolph.

5.5 Discussion and Conclusions

Some of the immediate consequences of the generative modeling approach have been discussed. But there are other consequences that may be less obvious. In any case it is certainly vital to reflect on what has been achieved so far, which limitations still exist, and of course whether the results achieved may also be useful for other purposes.

5.5.1 The GML as a Generalization of the Known Ways of 3D Modeling

Operator chaining as generalized modeling history. The *assembly line* metaphor for 3D modeling is supported by the fact that practically all of today's modeling systems use a modeling history. This concept is quite old, and it dates back to the first CAD systems in the early eighties. It is sometimes limited though to global operations. In 3D Studio Max from Kinetix Inc. the history is called the *modifier stack* [3DS]. 3D Studio permits to apply arbitrary deformations, twist, warp, and also subdivision to each object. The parameters of every operation can be changed afterwards, and the object immediately reflects the changes.

The GML generalizes the modifier stack and replaces it with the more general technique of operator chaining. It is superior, e.g., because it permits *branching* of the modeling history: Let toolA have not one but two results, let toolB process one of them, and let toolC recombine the two. Branching is as simple as `toolA toolB exch toolB toolC`. Second, it is superior because of the functional background: The sequence `toolA { toolB } map toolC` will work for result sets of any size if toolA and toolB are made to create and process a whole array of items at a time.

Generalized data flow networks. The distinguishing feature of the high-end modeling tool *houdini* from Side Effects Software Inc. is that it permits to define a *data flow network* for 3D modeling and animation [Hou]. In fact the whole software system is based on the concept of processing nodes where output connectors can be routed into input connectors. Models such as a procedural gear are possible in Houdini with its pre-defined looping constructs.

Another example where a data flow network approach has been quite successfully used is in the *werkkzeug* toolkit for procedural texture synthesis [FFCg]. It was produced by the *farbrausch* group that is very active in the German "*demo-scene*". This is a subversive movement of free non-commercial programmers who take part in an ongoing contest with the objective to obtain the best possible result with a very small executable computer program; usually the size of the binary is limited to 64 kilobytes. This task obviously requires heavy use of procedural techniques since almost no data can be stored within the executable. So these demos use unfolding information to the maximum.

An important contribution of the GML is that it is a generalization of data flow networks. A processing node in the network is nothing but an operator. Instead of connecting output to input via explicit routes the GML stack can be used for a much more organized and flexible parameter exchange; e.g., the stack can have any depth. Some data flow networks permit to collapse a whole sub-network into a single new processing node; in GML terms this is nothing but a combined operator. All capabilities of data flow networks can be mapped to GML constructs in a straightforward way, so the GML recommends itself as internal representation and as file format for data flow networks.

And GML is even more powerful: Functions can be parameters of other functions. This opens a new perspective on the capabilities of the GML. In network terms this means that the result of a processing node in a network can be a network; a network can also be the input for another node, so a whole network can in fact be transmitted over a network route; and finally a network can produce other networks as a result.

The software architecture conjecture for procedural 3D modeling. There are good reasons to argue that procedural modeling exploits only one half of the potential of procedural shape descriptions. The *software architecture conjecture* states that a consequent pursuit of procedural and generative modeling will always lead to a combined modeler/viewer software architecture, rather than to only a modeling software or just to a shape description formalism.

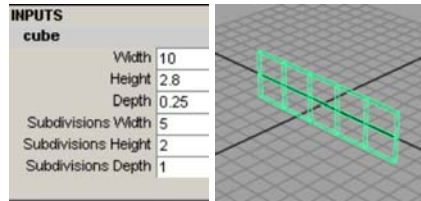
As already explained in section 5.1 most procedural modeling packages have a built-in scripting language; but unfortunately there is no common exchange standard for procedural models, and the deeper reason for this is that such an exchange makes only sense when the modeling tools are exchanged as well, on the binary level. But this is not realistic since then a software for viewing, e.g., CATIA and Maya models would have to incorporate the complete CATIA [Cat] and Maya [May] software packages.

The GML software architecture therefore marks a potential way out of this dilemma. Sooner or later the demand for rich online experiences, the *market pressure*, will become so strong that some sort of combined modeler/viewer architecture will be made available. Maybe an upcoming version of the *Playstation* from Sony or the *Gameboy* from Nintendo will have Maya or 3DStudio incorporated. But proprietary approaches have led all too often in the history of computer graphics to incompatible technologies and, thus, to a standstill. Hopefully the GML operator calculus is both simple and powerful enough to serve as general shape description language, and the GML engine as combined modeler/viewer architecture proves extensible enough, to foster the development of a free standard.


```

/object dict def
object begin
  /data 12 def
  /function { dup mul } def
end
object.data object.function → 144

```



```

Dialogue-Box begin
  /Width 10 def
  /Height 2.8 def
  /Depth 0.25 def
  /Subdiv (5,2,1) def
end

```

Figure 5.59: GML dictionary as object-oriented class. A dictionary can be used as object in the OOP sense, it may contain both data and functions (a). This permits to generalize dialogue boxes. (b): dialogue box for a subdivided box from 3DStudioMax. (c): Its potential representation as a GML dictionary.

GML dictionaries as generalized dialogue boxes. A dictionary containing only data and no functions can be used as a classical C-style ‘struct’. It also corresponds to a record in a database, and it can be understood as a form to interactively fill in entries. This is the concept of a *dialogue box*. They are frequently used in graphical user interfaces to further configure a specific action. They are also useful for describing objects, in which case they are known as *property lists*.

The GML can not only efficiently manage and store dialogue boxes and property lists in dictionaries. It opens also a novel perspective, namely properties containing executable code. Expressions can be used, variables can be referred to, and, most importantly, function calls can be issued from ordinary dialogue boxes, as suggested in Fig. 5.59.

Dictionaries can take on many roles, especially in conjunction with path names. Note that the latter also have much in common with the path names used in file systems; just replace the dot by a slash³. A dictionary hierarchy in turn has much in common with a hierarchical file system. Even hard links are possible, since a dictionary can be referred to by many other dictionaries. Rather than of a hierarchy one should therefore speak of a dictionary *network*, like the network of wall and pillar dictionaries of the cathedral (section 5.4.4). Note that there is no ‘.’ operator for dictionaries, but a root dictionary exists.

Intelligent 3D objects, also for internet transmission. Another way to look at parameterized procedural 3D objects is to see them as small little program snippets or applets that realize one element in a responsive 3D environment. This is sometimes referred to as an *intelligent 3D object* and used, e.g., in architectural house planning software. Door, stairways, roofs etc. are all obvious candidates for being elements from some 3D object library. The most important property of these object is that they can adapt to the requirements of a particular building, i.e., that they behave ‘intelligently’.

The GML might serve as exchange format for intelligent 3D components. An existing example of an intelligent GML object is the Gothic window: it adapts automatically to the width of the walls in the cathedral. A capability of the GML that goes even beyond using intelligent object is that intelligent objects can in fact be *created* at runtime to expand the object library.

5.5.2 Persistent Naming and the Picking Problem

An annoying conceptual limitation of the modifier stack method mentioned earlier is that under some circumstances the stack may need to be frozen. In 3DStudio Max, for instance, this is the case when a deformed primitive (cube, cone, NURBS patch etc.) is converted to a *mesh object*. The reason is that the meaning of a mesh modification is not clear at all when the mesh undergoes more radical changes. A vertex inserted by a beveling operation might disappear when the bevel parameters change. What to do with an operation later in the sequence that manipulates this vertex?

This is one form of a fundamental problem in 3D modeling, the *persistent naming problem*. Several aspects of this problem are concisely listed in the article from Hoffmann and Joan-Arinyo in the *Handbook of geometric modeling* [HJA02]. The problem is to assign a name or ID to a part of an object so that it can be uniquely identified also when the problem undergoes parameter changes, i.e., the ID should persist. Another incarnation of the persistent naming problem is the *picking problem*: Assume the user picks, e.g., stair step number 7 of a procedurally generated staircase with ten steps. Then what should the system do when the number of steps in the door changes? The system cannot guess the rule the user had in mind when selecting the stair step. Perhaps its persistent name is ‘step 7 from below’ or ‘step 4 from above’ or ‘step at 2m height’; in any case an explicit rule is needed for the system to re-generate the new ‘step 7’.

Persistent names in the GML: Do they persist? The GML has two answers to the persistent naming problem. The first is to avoid picking altogether during the construction of an object. Interactive picking is deemed so very intuitive; but in fact it is the end of all procedural reasoning whenever meaningful changes are made interactively but the rule behind

³or a back-slash for some

```

v -1 -1 -1      (-1,-1,-1) v      { usereg
v -1 +1 -1      (-1,+1,-1) v      !faces !points
v +1 -1 -1      (+1,-1,-1) v      :points { addVertex } forall
v +1 +1 -1      (+1,+1,-1) v      :faces { addFace } forall
v +0 +0 +1      (+0,+0,+1) v      } /create-IFS exch def

f 1 2 4 3      [ 1 2 4 3 ] f      [ (-1,-1,-1) (-1,1,-1)
f 1 3 5        [ 1 3 5 ] f      (+1,-1,-1) (+1,1,-1) (0,0,1) ]
f 3 4 5        [ 3 4 5 ] f      [ [ 1 3 5 ] [ 3 4 5 ]
f 4 2 5        [ 4 2 5 ] f      [ 4 2 5 ] [ 2 1 5 ] [ 1 2 4 3 ] ]
f 2 1 5        [ 2 1 5 ] f      create-IFS
    
```

Figure 5.60: Versatility of the Postscript syntax. A triangle pyramid as an indexed face set in .obj file format syntax (a) and how it translates to the GML (b) when v and f are functions. The similarity is obvious, but note the reversal of the order of keywords and arguments. (c) shows the formulation as re-usable function.

the mouse click is not made explicit. Picking was used in none of the GML examples so far. This is not a conceptual limitation (see 5.5.4). Instead the GML actually encourages the artist to formulate the construction in terms of rules, as it offers all the time to re-use (and re-arrange) construction steps from before.

Second, in a more concrete sense persistent identifiers are realized by the construction itself. One example are the *anchor edges* in the grid of pillars from the cathedral. They are persistent in the sense that they are in a defined position on the top or bottom of a pillar. But they are defined by the pillar generating function rather than by the user clicking an edge and saying 'this is an anchor'. Or, to put it differently, mesh halfedges never appear in the GML code of a construction as E19,2,23423 (although syntactically they could) but always as a variable, e.g., pillar.topedge or :edge1.

But the persistent naming problem can of course not be avoided altogether; it is fundamental, and to solve it completely would imply a solution to the problem of shape description. But part of the latter is a creative challenge, which makes it impossible to solve by the use of a computer. – The concrete form of the problem was already mentioned: Anchor edges are fine, but which are good anchor edges? The bottom edge of the front side of a wall is good as an anchor edge only as long as the wall has no door. In the door example in section 5.3.4 the bottom edge is even destroyed. So in the worst case it may be that *all* edges of the wall are replaced; in this case the only way to keep a reference to the wall is to make the wall manipulating functions to explicitly update the persistent anchor.

The dilemma is that in order to, e.g., define an edge to be a persistent anchor, one has to foresee all possible uses of a construction. No anchor edges have been left inside the decoration of the couronnement of the Gothic window. This was a deliberate decision because only the style functions are supposed to decorate the couronnement.

5.5.3 A new way to think about Shape? – Shape Understanding and Shape Complexity

The GML is designed as a general file format for 3D model exchange. In principle it can represent indexed face sets and triangle soups just as efficiently as NURBS patches, scene graphs and implicit functions, as proven in Figs. 5.60 and 5.61. So the GML can be used as generalized low-level shape representation. A practical advantage of the stack based approach is that a developer does not have to worry about writing a parser in order to support a particular 3D file format. Instead, all that needs to be done is to define an appropriate set of operators to extend the set of built-in GML operators. Experience shows that it is in many cases possible to transform a given ASCII file format directly into GML syntax, e.g., by using a stream editor.

The Kolmogorov complexity. The great benefit of a simple syntactic transformation into the GML notation is that all kinds of *regularity* in the data can be expressed in a concise way – at least in theory. As a very simple example consider the sequence of natural numbers from 1 to 10:

1 2 3 4 5 6 7 8 9 10

Obviously a much better way to represent this sequence is by means of a very short computer program. The advantage is not only a reduction in size, but also greater manipulability: By changing only a single data item, one piece of information, it is possible to generalize on the data. In GML notation, some variations read like this:

```

1 1 10 { } for  → 1 2 3 4 5 6 7 8 9 10
1 2 10 { } for  → 1 3 5 7 9
1 2 20 { } for  → 1 3 5 7 9 11 13 15 17 19
    
```

<pre> Transform { translation 0 8 0 children [Shape { appearance Appearance { material Material { diffuseColor 0 0.5 1 } } geometry Cylinder { height 18.0 radius 0.5 } }] } </pre>	<pre> Transform (0,8,0) translation [Shape Appearance Material (0,0.5,1) diffuseColor EndNode material EndNode appearance Cylinder 18.0 height 0.5 radius EndNode geometry EndNode] children EndNode </pre>
---	---

Figure 5.61: Versatility of the Postscript syntax. The left column shows a portion of a hierarchical scene graph in VRML syntax. It could be translated to GML by using functions for nodes and fields.

The size of the generative description is not in the order of the output, but of a smaller complexity class: Instead of $O(n)$ it is $O(1)$. This effect is well-known from information theory, and it is formalized in the *Kolmogorov complexity* of a given piece of information.

Definition 5.1 (Kolmogorov complexity)

Assume that a computer model or a computer language is given. Then the Kolmogorov complexity of a bit sequence is the length of the smallest computer program that produces this bit sequence.

The Kolmogorov complexity is helpful as theoretical device, but unfortunately there is no way to compute it practically. It is very difficult both to prove or disprove that a candidate program is the shortest possible program. It is clear though by theory that the Kolmogorov complexity increases with the 'randomness' of the data. This is consistent with intuition: If there is no regularity that permits to derive some of the data from other data then every bit must be noted down.

The Kolmogorov complexity forms the background of all (lossy or non-lossy) compression algorithms: Information is first transformed into a space where it can be ordered according to significance, then optionally the least important data are removed, and finally entropy minimizing encoding is used (zip). Fractal image compression or the JPEG scheme with its discrete cosine transform are perfect examples. In the Kolmogorov sense, JPEG and fractal compression are also good examples for the principle of information unfolding from section 5.1. So this principle is not new at all; and it was never claimed it was. Its purpose is only to serve as inspiring change of view.

Deciphering structure is key to apply the generative method. A generative shape description is most effective when (i) a shape is understood and (ii) this understanding can be mapped to a formal description, such as a GML program, to make it explicit. Unfortunately both steps are not trivial.

It is remarkable that when a human seeks to understand the structure of a 3D shape then he often proceeds by reverse engineering the object mentally. Typical questions are: How has this been made, which parts are similar, how does it fit together, is there a reason why this measure is so, which material was used, and which manufacturing method?

Every answer that can be found to questions like these helps to increase the regularity of a shape. With an increasing portion of the shape becoming amenable to rule-based (re-)generation the amount of 'randomness', in the Kolmogorov sense, becomes smaller and smaller. But it is important to realize that for principal reasons this process is not static or automatic. There is no hope to find any finite list of questions about a 3D object that, when answered, deliver a perfect generative model. Again arguing with the human perception, a human deciphering shape works not only deductively, but also inductively, by partly 'inventing' the meaning of a shape: In the process of deciphering the structure of a 3D shape a human in most cases also seeks to *generalize* the construction. So the result is not necessarily the smallest possible shape description, i.e., optimal in the Kolmogorov sense. The size reduction may be considerable, but much more in focus is a decomposition into manageable, re-usable, and understandable components.

Intrinsic and extrinsic shape parameters. The Kolmogorov complexity is defined with respect to a programming language. In other words the decomposition of a shape into components depends also on the available shape operators. Different sets of shape operators may be formally equivalent because can more or less describe or approximate the same



Model	ascii	zipped	tokens	dict	func	Eulerops	V	E	F	$\Delta_{\text{depth } 0}$	$\Delta_{\text{depth } 4}$
Cathedral	131667	18612	10040	17	313	65967	36568	121706	25407	75357	7078485
Window	61860	10261	7018	18	282	29179	16244	57800	12798	32912	6811874
Chateau	32664	4999	1152	1	45	21820	11299	37548	7485	22542	3590473
Rosette	3045	1175	511	0	25	4490	2406	8952	2098	4812	1079100
Gear	1527	592	248	0	4	1030	528	1992	464	1080	223740
Towers	1037	506	117	0	3	908	470	1808	438	932	197820

Figure 5.62: Comparison of the different complexity classes. GML ascii and token size, Euler sequence, mesh vertices, edges, and faces, and triangles. Note how large the range of triangle sizes is that can be produces.

surfaces. But the question whether a given set of operators (and the shape representation it operates on) is suitable for generative modeling also depends on the mental process described before. The decomposition into block structures facilitates the process of describing contemporary architecture; but blocks are less applicable for the Gothic style, or for describing automobiles. Therefore not only the shape description itself, but also the underlying shape representation need to be chosen according to the domain.

More formally, what a shape operator set must facilitate is the re-parametrization of shape, i.e., to convert back and forth between *extrinsic* and *intrinsic* parameters. Intrinsic parameters are measures that are not free but determined by the construction. The height of a square equals its width; and the length of the diagonal is also only a consequence. So the square has only one externally specifiable, or extrinsic, parameter. But the rectangle has two, its width and its height. Important is the ability to convert one to the other: A rectangle is specialized to a square by converting one extrinsic to an intrinsic parameter, i.e., to bind it with a determining rule. The other direction is equally important: To generalize a square to a rectangle by freeing one of its intrinsic parameters. The inductive method prescribes to first turn the one-parameter square into the two-parameter rectangle to then re-use the rectangle to define the square by parameter binding.

A shape representation with a set of shape operators that facilitate this conversion is well suited for generative design.

Automatic conversion and generative shape acquisition. When presenting the GML as a new low-level file format for 3D shapes the most common question is: “Is it possible to convert models from format X to the GML format?” – The answer to this question is both yes and no.

Formally the answer is of course **yes**: Every primitive based format can be expressed in GML syntax, and this applies also to more complicated and hierarchical formats as suggested by Figs. 5.60 and 5.61. But it is important to understand that this will unfortunately not solve any of the inevitable problems of shape descriptions based on lists of primitive objects. A conversion to GML syntax alone does not lift a shape on a higher semantic level. But it is nevertheless favorable since it permits for a gradual transition to exploit more and more of the shape regularities.

Then evenly spaced points, for instance, can at least be generated using a loop.

A new measure for shape complexity. The size of a token is 16 bytes. This is the same as four 32 bit IEEE floating point numbers or a 3D vector in homogeneous coordinates. As a remark, 3D vertices are on the hardware and driver level most often represented with four rather than three components, to align the memory to 128 bits for fast vertex access.

The fact that a 4D vertex and a GML token have the same sizes also permits to compare the sizes of generative models and triangle meshes, as carried out in Fig. 5.62. The cathedral has only three times more vertices (V) than tokens. But note that only the simplest window style is used in the cathedral; imagine some variations of the window from 5.62 (b) were used for each of the 78 cathedral windows. This would increase the size of the generative description only negibly, by the size of few window dictionaries. But the size of the respective triangle mesh would explode. Since the windows are different, reference instances would not help either. But of course the shape of the cathedral does not become more complex from an information theoretical point of view only because some of the shape parameters are changing.

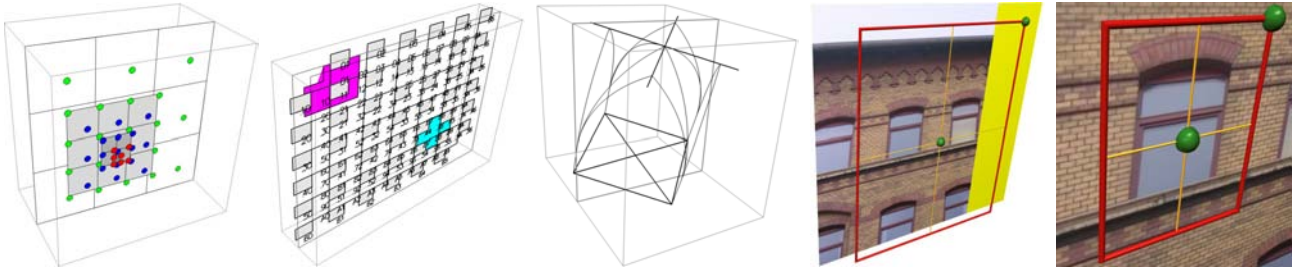


Figure 5.63: The GML as a tool for creating diagrams and interactive applets. **All** diagrams in this thesis have been created using the *XFig-extension* of the GML. Examples (a), (b), (c) are for Figs. 3.33, 3.27, and 5.23. As diagram elements are allowed to overlap (in Xfig and PostScript) diagrams are $2\frac{1}{2}$ D rather than 2D - so why not use 3D directly? (d,e): Interactive GML applet to crop the images for Fig. 1.3, 1.5, and 5.1

The GML permits a gradual transition from a primitive-based to a generative shape description, which makes it compatible to both worlds. It also permits to compare different approaches for describing the same shape. So we would like to propose *the number of GML tokens needed to a shape as a new general measure for shape complexity*, instead of the less relevant triangle count.

5.5.4 Extending and Embedding the GML

All images and diagrams in this thesis have been exclusively generated with the GML⁴. This underlines again the power of generative modeling: Many Figures in this thesis attempt at illustrating different aspects of the same thing. With only a single image it is not so clear where to direct the focus of attention. A series of images can more distinctly lead to the interesting aspect if this aspect is varied each time a little bit. This led to the idea of using *serial images*. But to generate a series is obviously much easier when only a few high-level parameters have to be changed from shot to shot.

On the origin of the drawings in this thesis. To use the GML for the drawings was due to a very practical reason. The first attempts with vector drawing programs such as Xfig and CorelDraw were disappointing since, e.g., intersections could only be approximated. In fact in many cases the whole ‘true’ drawing could only be approximated. To use AutoCAD appeared too cumbersome too: despite its very efficient drawing tools it is not easy to create serial images with it. So the final choice was to use Xfig, but only as a file format and not as a drawing program. The file format of Xfig is pure ascii, extremely concise [SSS*02], and exporters from Xfig to PostScript exist. It may seem a little bit artificial to let the GML generate Xfig drawings that are then converted to PostScript in order to include them in a LaTeX text. But this was in fact the fastest way to achieve a good diagram facility, and the drawing elements of Xfig are sufficiently powerful.

An image cropping applet. Even some of the foto series were created through using the GML. It is extremely cumbersome to crop and resize as many images as needed, e.g., for Fig. 1.3 by using an image processing tool. In this particular type of tabular the heights and widths of the rows and columns can all be different. So each image can have an individual aspect ratio. Many slight variations based on a simple rule – a great case for the GML.

An interactive image cropping applet was created where the images were used as OpenGL textures for the mesh, and a frame could be moved interactively over the image. Note that there are four free DOFs: the position (x, y) of the center and the scaling (s_x, s_y) . These, and only these, DOFs can be manipulated by moving the balls in Fig. 5.63 (e) by interactive click-and-drag using the mouse. In this case the use of the mouse is justified: No automatic rule can be devised for cropping the right portion of the image, and each image shall be treated individually.

Interactive GML applets. So far the GML has only been introduced as a formal language for representing generative shape descriptions. But originally one of the main incentives for developing the GML was the problem of truly rich interactivity. This aspect will be an important focus of future papers and reports. A brief idea of how it works is given by the image cropping applet and the *interactive CAVE designer* in Fig. 5.65. Image (a) shows balls and arrows that can be interactively dragged into the arrow direction using the mouse. Such elements that are not really part of the scene, but just stand for operations, are called *gizmos*. In particular the whole CAVE frame can be moved with them, and all derived data, e.g., the size of the mirrors necessary for redirecting the projected image, are instantly updated online. The interactively draggable balls, sticks, and arrows are part of the *BnS-extension* (for **ball and stick**) of the GML.

⁴except the historic images explicitly labeled to come from other sources, and the fotos from Braunschweig and Cologne in section 5.4

```

class Token {
  short v_type;
  short v_status;
  union {
    int v_int [3];
    float v_float [3];
  };
  ...
};

class GMLop
{
public:
  virtual ~GMLop () {}
  virtual string& name () const = 0;
  virtual GMLop* create () const = 0;
  virtual bool init (GMLInterpreter&);
  virtual bool execute (GMLInterpreter&) = 0;
};

class GMLopCross : public GMLop
{
public:
  GMLopPrelude (Cross, cross);

  virtual bool execute (GMLInterpreter& inter) {
    Token* v1 = inter.pop_stack();
    Token* v0 = inter.pop_stack_leave();

    if (inter.error() || !(v0->isP3() && v1->isP3())) {
      return inter.setError(GMLInterpreter::TypeError);
    }

    v0->set(v0->asP3().cross(v1->asP3()));
    inter.commit_pop();

    return true;
  }
};

```

Figure 5.64: GML implementation.

The GML implementation. This was possible only because the implementation of the GML is extremely concise. The GML interpreter comprises just a single implementation and header file that currently contains only about 90 KB of C++ code. Two of the main ingredients are shown in Fig. 5.64: The Token class and the pure virtual base class GMLop to derive GML operators from. The implementation of the cross product operator is shown to the right. Most of the virtual functions are defined by the GMLopPrelude macro, so that the programmer can focus on implementing the execute method. Many GML operators have very short implementation because they are just wrapped library functions like cross.

The next goal is to make the GML interpreter and its runtime engine *open source* so that they are freely available. This requires much work to polish the code, though. Many of its features are therefore subject of future reports: Generating GML ascii source code from tokens alone; GML resources to extend the type set; resource applets to integrate gizmos and new shape representations; the callback concept for interactive modeling even in distributed environments; and the simplicity of embedding the GML. This will reveal how applications can be automatized to gain unprecedented flexibility, like with the following simple replacement of a typical code line used to enter a floating-point value:

```

float height = atof(textbox->text());    simply becomes
Token code;
interpreter->parse(code, textbox->text());
interpreter->call (code);
Token result = interpreter->pop_stack();
Float height = result.asFloat();

```

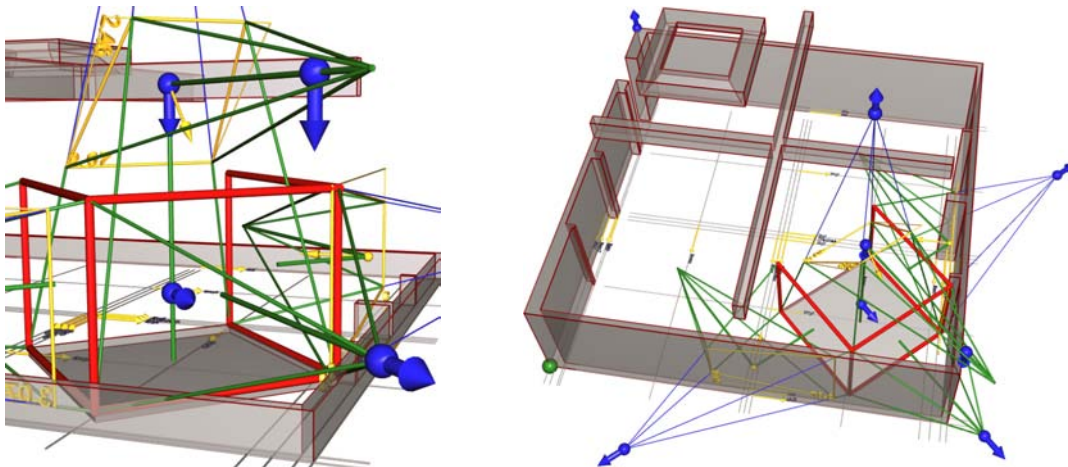


Figure 5.65: The interactive CAVE planner, a useful GML applet.

5.6 Future Work – Fields of Application for GML-based Technology

Each and every new computer sold today offers thrilling powerful hardware support for 3D. This development was stimulated by the increasing commercial significance of computer games. But as enjoyable as they are, the use of 3D should not stop with games as the ultimate *3D killer application*. The crucial – open – question is how the undeniable potential of 3D can be made available to all different areas where computer technology is applied. The technical prerequisites for *3D everywhere* and *3D for the masses* are there – but how can the vision become reality?

Some advantages of the GML have already been mentioned, but the true potential of the GML approach may go far beyond that. Some concrete and some more far-fetched – *and maybe rather speculative* – ideas are sketched in the following. The central hypothesis is that this is possible only with a radically different type of 3D software technology. Technically, the key to ‘3D everywhere’ is the *integration of 3D modeling capabilities* into applications or, even better, into the OS.

5.6.1 3D Objects for Everybody

Problem. Creating 3D objects is too cumbersome. Today’s modelers all work by providing a (possibly large) set of primitive objects (triangles, NURBS, spheres etc) that can be manipulated in arbitrary ways. Most users however just do not wish to design objects from scratch. They would rather take an existing object with well-defined functionality and configure it according to their needs and wishes. – Consider a scenario where an individual or a company is assembling a 3D scene, for whatever purpose, where a complicated but generic object is required, for instance a bridge. Probably no bridge that can be found, e.g., in the internet will perfectly fit: it is either too long or too high, or its style is inappropriate; and bridges cannot simply be scaled. – Examples of this type show that a wider spread use of 3D is hard to imagine without a great variety of intelligent, highly customizable 3D objects available.

Solution. The GML is not only the ideal device for describing parameterized objects, but it also supports most flexible forms of customization. Objects can not only be parameterized in terms of floating-point parameters, but also in terms of the functions for particular features. If taken to the extreme, this makes it possible to define 3D objects in a completely abstract way. Examples like a bridge are a hierarchical combination of procedural features. The choice of the bridge type for instance can be further differentiated by a variety of options such as whether to use bars or strings for connections etc. An *abstract 3D object* is then only a set of interface definitions or pre/post conditions of the different features. Pre-defined simple instances of the features may serve as default, more complex features can be plugged in on demand, very much like the Gothic window from section 5.4.1 with its different styles and style combinations.

A new market for trading procedural objects may result, with companies or gifted individuals offering intelligent customizable 3D objects commercially for download.

5.6.2 A Variety of Operator-based Shape Representations

Problem. Not all shape types can be equally well represented by combined B-Reps. They were designed as a reasonable ‘work-horse’ that combines the flexibility and generality of meshes with a reduction of degrees of freedom for free-form parts of the surface. But the guiding idea behind them is information reduction and the principle of unfolding information: Shapes with fewer degrees of freedom are more manageable and more easily ‘understandable’ and, thus, better suited for procedural parametric design. But for more specific classes of 3D objects other representations from the large ‘zoo’ of shape representations in computer graphics may be more suitable – which means that for certain domains other representations are much more manageable than combined B-reps.

Solution. The GML is by no means restricted or specially tied to combined B-reps. To integrate any other shape representation is just a matter of finding a suitable set of operators to create, modify, and delete parts of a shape. Metaballs for example, a type of implicit surfaces, can be concisely described by a pair of operators: (x,y,z) radius **create-ball** → Metaball to create a metaball and another operator Metaball Metaball **blend** → Metaball to set up a tree of blended shapes with balls in their leafs and blend operations as inner nodes.

The right set of operators creates a ‘shape calculus’ that is both sufficient and closed with respect to the chosen domain.

Another beautiful example are *brushes*, i.e., convex solids created as the intersection of half spaces. For their very nice computational properties, brushes are the preferred building block of the thrilling virtual worlds in game engines (Quake [Qua], Halflife [Hal], etc.).

It will be both intellectually challenging and scientifically rewarding to systematically review the large body of literature on shape representations in order to discover the right set of operators for each.

5.6.3 GML + OpenSG to replace VRML/X3D: Scene Graph Scripting with Lazy Evaluation

Problem. VRML is not a modeling language, no fine-level mesh manipulation is possible with indexed face sets. Adaptive LOD requires to keep the highest resolution in memory. The only advantage of VRML over a flat list of primitives is that it is *hierarchical*. But this does not provide any more higher-level semantics: The internal parameters of the objects in the scene graph are not explicit and, thus, in no explicit relation to each other. The export from a modeler to VRML is a dead end because it breaks the link to the modeling history; there is no way to store it with the exported model. To provide VRML models with interesting behaviour requires not only to master an external modeler for object creation, but also literal programming in Java or JavaScript for the behaviour. Both commercial artists and programmers are very expensive.

Solution. Our group was active in the OpenSGplus project, a collaborative effort of computer graphics groups in Germany to extend the open source scene graph engine *OpenSG* [RVB02, Opeb, Opec]. It supports VRML as input format but has no scripting engine yet. A scene graph is a graph after all, though, and the GML was already shown to be efficient for creating (cB-rep) meshes. Also GML is good for 'emulating' VRML (Fig. 5.61), but it can do even better.

First, the scene graph itself, rather than only fields of scene graph nodes, can be dynamically changed. Nodes and whole sub-trees can be added, removed, moved from one place to the other etc. On the C++ level OpenSG has a similar mechanism as VRML, with fields and *FieldContainers* attached to nodes and 'node cores'. The GML in turn has a facility to make custom types, added by a GML resource, behave like a GML dictionary. By combining both features it is possible to use path expressions to navigate conveniently through the scene graph to change and access it.

The most fascinating perspective is a scene graph with *lazy evaluation*: A GML node as OpenSG custom node type might contain GML code that creates geometry only on demand. When the node is invisible, the geometry is *completely erased* from memory. When it is needed again it can simply be re-generated from scratch. With this mechanism a complicated, detailed piece of VR furniture might only be created in the moment when the user enters the room – and destroyed when he leaves. – Björn Gerth is pursuing this idea in his diploma thesis. When the ensemble works then GML+OpenSG will be a serious alternative to VRML. Unlike VRML the standard will not just comprise a (sloppy) specification but also offer an open source reference implementation.

5.6.4 Computer Games about Creativity rather than Destruction

Problem. A great problem with many current computer games is that they are simply not interesting for many people, in particular women, who do not want to waste their time with gaming. The majority of all 3D games is about destruction, and they basically train the user's reflexes (and stress resistance) – typical genres range from ego-shooters over war simulations to space combats; also many racing games fall into this category. Much fewer games are about construction, and those are mainly from the simulation genre. And only very few games have their focus on creativity – which is at the heart of the way children for instance play.

Solution. Abstractly spoken the GML permits to define "processes" at runtime. In some sense to create a GML program is very much like playing with Lego – only plugging compatible operators together rather than compatible plastic bricks. But the Lego analogy goes even further, as the GML is about 3D. So with GML based technology it is also possible to play literally with Lego on a computer, provided there is a Lego library – or bricks, or *Fischertechnik*, or *Märklin*, or *Playmobil*, just to name a few "system toys" known in Germany. The advantages of using GML are (i) of course unlimited resources (of bricks) and (ii) the possibility to define functions for automatizing cumbersome sub-tasks (building walls).

5.6.5 A Double Layered Market for 3D Components

Problem. It appears that there is practically no market today for trading 3D objects. This is due to the already mentioned problems of limited *changeability* and *re-usability* of the static 3D models. It would only make sense to trade intelligent components, but high-end parametric technology suffers from incompatible proprietary formats, as already discussed: An intelligent 3D object in Pro/Engineer is useful only for Pro/Engineer customers. – And IPR protection is an issue.

Solution. GML based technology might be the catalyst for a new 3D market. First a market for binary components, dynamically linked custom operator libraries (.dll,.so), to extend the built-in operator set (also see section 5.6.10 and the software architecture conjecture from section 5.5.1). Binary components are also a way to protect intellectual property, to prevent competitors from stealing the procedural knowledge that, when coded in GML, is openly readable.

The second layer market is about trading GML components: Today an engineering office told to develop a specific new machine would hand in the plans. In the future it might develop not only a single machine, but a customizable plan for a machine, i.e., a parameterized manifold of machines instead. – And also *recreational modeling* might become hip.

5.6.6 3D Support for all Computer Applications

Problem. All computer applications that deal with real-world data are potential candidates for integrating a 3D visualization. Any management of real entities might at some point need to display the real-world location of the items, or the spatial relation between them. Many types of data can be efficiently represented in a tabular – but if the tabular becomes too large, a different, more abstract, presentation is more convenient. Variations in the data must be visualized with variations in the 3D objects for which an application requires built-in *modeling capabilities*. At some point it is not sufficient to be able only to scale and rotate 3D objects that are externally created and imported; instead the objects must be created on the fly. – Moreover 3D visualization should not be a one-way road. If the assumption is true that some data are much more comprehensible when shown in 3D, then it is also true that it is more convenient to manipulate these data in 3D.

Solution. The GML is designed as an embeddable scripting language. To provide an application with a GML-based 3D visualization is as easy as to create a GML interpreter and an OpenGL canvas that the GML can render into. The interpreter can either load and execute pre-defined libraries, or parse and execute GML code from dynamically assembled character strings. Both options can also be combined. The descriptions of the parametric objects that are to be created dynamically are stored in a GML library, and the actual creation of the objects is (in the simplest case) triggered by a dynamically generated string that contains the current set of parameters.

```
gmlInterpreter→parseAndExecute("(10,7,0) 3 12.0 HouseLibrary.create-simple-house");
```

Care was taken that new operators can be created easily, which is especially useful for *application-specific operators*. This makes it possible to export some of the functionality of an application to the scripting language. This provides the way from the 3D visualization back to the application when GML callbacks trigger the execution of application-specific operators. – The GML was already ported to many skins: ftk, MFC, Qt, ActiveX, Firefox, and Aqua (MacOS X).

5.6.7 True Generalized 3D Documents

Problem. The convergence between conventional forms of textual documents, such as the classical book, and multimedia content – also called *generalized documents* – was one of the driving forces of the world wide web. There were significant developments on both ends of the spectrum to come closer:

- **The classical print industry** was revolutionized by the use of computers. The whole workflow is digital today, from the author using a word processor to the physical book, printed on demand, on the end of the chain. Authors can produce ready-to-print digital documents containing text, figures, diagrams, and photos, and deliver them to the publishing house or directly to the printer. With digital delivery, readers can even directly download books as PDF files and print them at home – or read them as electronic books on a portable computer.
- **Classical menu-driven computer applications** more and more take the form of an electronic document. Almost everything can be ordered online using html forms, using the web browser as a door to highly complex server-based computer applications – with menus, forms, and a complete graphical user interface. Parts of the server application can even be transferred to the client using Java applets, or with JavaScript embedded in html.

This thesis can be printed to read it, or it can be read as an electronic document. The latter provides more functionality: All references to sections and figures are hyperlinks, which makes it much easier to navigate through it. But concerning the figures, note that many of them contain series of similar images to make things clearer. Take for example the construction of the quad torus using Euler operators (Fig. 2.15, 3D) or the evaluation of a B-spline curve (Fig. 3.1, 2D). How much more instructive were such images and diagrams if the user was allowed to manipulate some of the parameters directly.

Solution. Didactic applets are a great example for the principle of information reduction: The reader may influence just a small number of high-level parameters and inspect a complicated model derived from these parameters.

A GML web browser plugin already exists. Even more natural would be the integration into the PDF format since PDF is derived from PostScript just as the GML is. But whereas PostScript is a pure output format, the GML shows that a stack-based language also efficiently supports interactivity. This is useful not only for embedding live diagrams, but also for serious purposes: A complete spreadsheet with $m \times n$ rows and columns could easily be exported to a PDF-GML file in a way that its functionality is fully preserved – provided that all spreadsheet functions are available as operators.

To maximize the benefit for the users, the GML should be reliably and tightly integrated with PDF by extending the standard, rather than by a plugin for the PDF viewer. This solution is also indicated by the similarity of both approaches. As a historical remark, the GML shows that PostScript has been completely under-rated as scripting language: The PDF format has only recently been extended to support forms that can be filled out interactively.

Concerning the size of PDF files note that all the images and diagrams in this thesis were produced using less than 100 kilobytes of GML code (compressed) – compare this to the size of the PDF file.

5.6.8 The 3D Desktop

Problem. Currently one major development in the operating system market is to provide the user frontend, which manages windows and determines the look-and-feel, with 3D support.

A 3D desktop may at first resemble a computer game. But advanced visualization concepts can provide very concrete advantages to make work more efficient: Smooth animated transitions instead of abrupt changes help to keep track of many windows; non-rectangular windows can save screen real estate; transparent windows show more than one layer of information; and a smoothly scaled down application window still providing live output is far superior to static icons. The shift towards 3D desktop was most notably triggered by the *Aqua* system, the user interface of Apple's MacOS X [App04]; *Looking Glass* is an OpenGL-based window manager for Linux on top of X Windows [KBJ04]. For Microsoft Windows the successor of Windows XP, called *Longhorn*, supposedly uses an advanced windowing system that makes use of the 3D graphics hardware available on every PC today.

The desktop is gradually moving towards 3D – but it seems like a long way to replace the familiar 2D by a true 3D desktop where applications are represented by objects floating in 3-space. One obstacle on the technical level is that the window manager must have built-in modeling capabilities: When the frame of a 2D window is a 3D object, then this must be a parameterized object; resizing a window does not mean to scale it. So static 3D models exported from a 3D modeler are basically useless; *live models* are required instead.

Solution. A true 3D window manager requires nothing less than a scene graph with a procedural parametric modeling engine. As pointed out before, GML based technology is very well suited for creating 3D objects on demand. But for a full 3D window manager, this is not enough. The real-time functionality of the GML would have to be considerably extended to be much subtle and permit to handle also low-level resources such as bitmaps and fonts.

A great number of different graphical objects or widgets (user interface elements) is necessary for a full 3D window manager: Menus, buttons, sliders, icons, panels etc. For a window system, usability is key. More importantly, the 3D components need to behave intelligently. Again the information reduction principle is key: Event-driven state changes in GML user interface objects may trigger the execution of callback operators from the underlying system implementing the user interface logics; and this system may in turn examine the high-level parameters of the GML widgets. – Today the look-and-feel of user interfaces is often customizable using 'themes' or 'skins'. Such an option nicely corresponds to the concept of style libraries from the GML.

5.6.9 New forms of Human-Computer Interaction

Problem. A great number of new, low-cost interface devices for human-computer interaction are being developed and will be available in near future. Inexpensive video cameras and high-resolution digital photo cameras in conjunction with photogrammetric methods make the computer much more aware of its surroundings than today. Camera-based tracking is already possible today even on the desktop. A 2D device like a mouse is sufficient for working in planar sections of 3D objects and for selection tasks even in a 3D environment. But for 3D object manipulation 2D devices are felt by most users as being insufficient for tasks such as moving or placing objects somewhere in space. This makes 3D input devices mandatory at some point.

On the output side, stereoscopic (augmented) and immersive imagery can give the illusion of true spatial depth, especially when combined with head tracking (eye positions). But the projected illusion becomes apparent with the absence of physical barriers – force feedback is not and will not be practical in the near future. So all physical properties and events of the virtual world have to be transported via sophisticated graphical means, such as colliding objects changing colors or spreading particles, or heavy objects moving with some inertia. Such 'soft' factors are often critical since they make the difference between success and failure – especially with 3D, swift usability is key. The many different combinations of input and output devices requires extremely configurable software.

Solution. Consider a scenario where the same application is to be used in two completely different settings: The first is as a desktop application with relative 6-DOF device (spacemouse, 10 buttons) together with a conventional 2D mouse as input devices and static stereo display (i.e., without head tracking). Literally the same executable shall work in a 4-sided CAVE system with 6-DOF game controller (two 2-axis joysticks, many buttons) and head tracking, which work both in an absolute coordinate frame. It is not sufficient to configure a device only on the device driver level; also the graphical user interface has to adopt to a different handling.

When changing input and output devices, a flexible reparametrization is again key. It must be possible to route the specific device events ('x-value +5') flexibly to more abstract application events ('move/rotate right', 'undo', 'twist object'). But abstract events alone are not sufficient since the reparametrization must also be reconfigurable dynamically: The navigation in a CAVE is fundamentally different from the navigation on the desktop.

5.6.10 Operator-based component technology

Problem. With existing technology it is very difficult for average users to define simple procedures and processes without programming. Every programming language, however, has its tweaks and complications. There is not much difference between learning a programming language and a foreign language: It takes time, one or two years, until a person can express him- or herself elegantly, express exactly what he or she wants to say, and avoid misunderstandings.

Component based technology was deemed to be the solution: Highly functional software components with well-defined interfaces could be glued together by average users using ‘wizards’ to make up custom-tailored applications. The great problem is that with most technologies, component interfaces are mostly static: A list of properties, basically a form that can be filled, where every entry can take a numeric or string value. The component is sent a message whenever any of the values in the form is to be changed, and can react on this event. The underlying model of this approach is basically a static data flow network: There is a strict distinction between the program (components) on one side and data (property forms) on the other.

Solution. A GML based component technology would be a true generalization of the existing technology in the sense that the first can emulate the latter. But GML based components can also offer new functionality beyond this, the reason being that it abolishes the strict distinction between data and operations. Software components with a GML operator interface can be glued together much more flexibly. As outlined before, a property form of a software component nicely corresponds to a GML dictionary. Events and messages to inform the component about changes in the form correspond to callback functions. – In a GML based component interface it is possible to store equally concrete values, expressions, function calls, and whole programs, such as branches and iterative computations. The width of a button may be set at runtime to "67" as easily as to "x 34 mul" – where "x" may be a variable as well as a function call. It may even change its meaning from time to time, since name lookup depends on the state of the GML dictionary stack (Fig. 5.5, rule 7).

This feature unfolds its full potential in conjunction with another, automatic code generation. Property lists are of course supposed to be filled out by software and not by a human – imagine every dialogue box in every program could be made to show only the relevant entries, complete sets of dialogue boxes could be stored and retrieved, etc. A new generation of flexible user support tools may then indeed generate GML descriptions of complicated applications from a ‘few mouse clicks’. The reason why this is realistic is again the possibility to use customizable procedural descriptions – yet unlike in 5.6.1 not only to describe the construction of 3D objects, but to describe general applications assembled from a set of software modules.

5.6.11 Integrating the GML into the Operating System Kernel

Problem. The *procedural modeling software architecture conjecture* from section 5.5.1 claims that consequently pursuing the procedural/generative approach must lead to an integrated modeler/viewer architecture. The 3D modeler needs to be called whenever the viewer decides. The GML operator calculus is the glue in between, as the modeling requests are formulated in terms of GML functions. But in fact both modules interoperate on the *binary* level.

The previously proposed component technology (section 5.6.10) goes one step beyond that. It requires not only modeler and viewer, but ultimately all software running on a computer to be interoperable in the same way: Each package may be used by each other package.

This may lead to security problems. The GML glues together extension modules in binary form rather than, e.g., Java bytecode. So GML modules do not run in a ‘sandbox’ type of secure environment. The consequence is that users will hesitate to download binary extension modules. And this, in turn, will be an obstacle to the dissemination of extension modules. Unfortunately the sandbox approach will not work for the GML: Interactive 3D visualization is quite demanding in terms of computing resources, so the additional overhead is not tolerable.

Solution. The security problems can only be reliably solved with an integration of the component technology into the kernel. An onion-like security policy can provide a downloaded module m with a *privileges*. The privilege determines which OS services m may use (technically, which libraries m may be linked against). In the 3D context an expert might have created a highly efficient GML module for interactive CSG on combined B-reps. This module does not need access to the file system, no connection to the internet, etc. – So it is perfectly content with a low security privilege.

A GML interpreter in the OS kernel would bring the additional benefit that a system call `gml("do some thing");` would be easy to use as `printf("do some thing");`. Also note that GML tokens are low-level friendly: Tokens are like 4-component float vectors, their size is fixed 4×4 bytes = 128 bits. So a GML program, a token sequence, can be stored, e.g., in an OpenGL vertex array. So token sequences could be used as *stored procedures* for message passing in the kernel.

And this might also lead to a generalization of unix-style piping of `stdin/stdout` from one application to the next: A whole stack of arguments can be piped rather than just a single object.

5.6.12 GML is stronger than XML

Problem. XML provides a possibility to describe arbitrary kinds of data in a structured way [XML]. The underlying organization scheme is a strict tree, written down in bracketed infix notation: <node> children .. children </node>. The great problem is that XML is too ‘modest’ in that it does not specify what to *do* with the data in the tree. Complicated extensions enable to break up the strict tree structure, and to validate or transform an XML document using DTD and XSLT [XSL]. These extensions are complicated, since they require a limited amount of procedural elements (such as if-clauses) which are not inherently part of the basic formalism: An XSL transformation can be written up in XML notation, but it requires an *interpretation* of the content – whereas XML only specifies a syntactic notation, and not an interpretation (which is even cited as one of the strengths of XML!).

Very simple things are not possible with the fundamental restrictions of this type of technology. Consider a simple scenario: A user has an (assume secure) E-mail program, an application for tracking personal finances (like Quicken, GnuCash), and another official program from the state for doing his tax declaration, an (also assume secure) electronic tax form (Elster in Germany, [Els]).

What the user wants is that the account receipts he receives regularly from his bank via E-mail are automatically sorted into the right ‘bins’ of his personal finance tracker, and that some (probably not all) of the bins are then summed and entered correctly to the right places in the tax form, which is eventually sent in time to the server of the tax authority.

Now also assume all these programs have XML in- and output facilities, and they can run in batch mode, i.e., non-interactively, as software components. But does that help? Probably not. The problem is that XML does not specify what to do with the data, where to put them, and how the XML output from one program becomes the XML input of another. One must resort to a second technology, XSLT, to transform the output document into an input document for the next stage. But then, one is again confronted with the limited procedural capabilities of the XSLT standard. In order to do simple things like sorting or summing up rows or columns of values, it is eventually inevitable to use a third technology: Programming, probably by using some Java or JavaScript hooks of XSLT. In by far most of the cases, it would be possible to get around programming if something like a spreadsheet component was available that can transform an input spreadsheet into an output spreadsheet; probably with a database enhancement using SQL [SQL91, sql96] to sort and re-arrange the records from the input XML-‘database’ to the output XML-‘database’.

Solution. This weird, complicated situation might possibly be resolved in a very simple way: Instead of introducing step by step different more powerful technologies, enhance the data format so that it can contain data processing instructions along with the data. As already mentioned, this is exactly what the GML provides; and again, it provides a generalization of an existing technology, because it can not only do the same, but more.

This is not only true on an abstract level; it may be surprising that there is a remarkably simple transformation from XML to GML. Line 1 of the following three lines is XML syntax (HTML), line 3 is GML, but note the line in between:

```
1 <b> this is bold <i> and italic text </i> just bold again </b>
2 b >this is bold< i >and italic text< ni >just bold again< nb
3 b "this_is_bold" i "and_italic_text" ni "just_bold_again" nb
```

Line 2 can be considered a GML program if the tokenizer is made to accept > and < as symbols that open and close a character string. Leaving away the first ‘<’ and the terminating ‘>’ character just reverses the notion of normal text: Instead of structured formatting symbols inserted into normal text, the document is now made of character strings that are arguments to operators. Yet syntactically, there is not much difference. And note that this is just what an XML parser does, since each chunk of consecutive characters is placed into its own text node in the DOM tree.

For the example above to actually work it is just necessary to provide operators such as b and nb to switch the ‘bold’ property on and off, as a side effect to consuming the text. The next example shows that when using GML, it is not necessary to resort to a second technology (e.g., DTDs) just to be able to define styles concisely. Lines 1 and 2 show the XML and GML versions just like above, and line 3 does the same as line 2 but using a style definition.

```
1 <font size="+1" color="red"> this is larger red text </font>
2 pushfont +1 fontsize "red" fontcolor > this is larger red text < fontpop
3 /fontA { pushfont +1 fontsize "red" fontcolor } def fontA > this is larger red text < fontpop
```

When the style myfont1 is declared in the preamble, it can simply be used throughout the whole document, which makes it much easier to change its definition. This ‘preamble’ technique is used very much in PostScript.

The same technique also helps to finally resolve the ‘automatic tax’ scenario. Assume the bank has defined a standard GML markup receipt .. end-receipt (instead of <receipt>..</receipt>) for all bank account receipt e-mails it sends. The e-mail program was told to ‘execute’ each arriving account e-mail – but before that it has to execute a special user-defined header to properly define and overload the default receipt / end-receipt functions. The end-receipt function can then decide where to sort a receipt, and call the appropriate data base call-back function; and end-message triggers the summing.

5.7 Epilogue

During the course of this thesis many fruitful and quite controversial discussions took place, especially with members of the faculty of architecture. The efforts for studying ancient style systems, such as Gothic architecture, has sometimes been disregarded as being anachronistic, and of only historic value, if at most. But to tell the truth, the initial motivation for studying the Gothic construction principles was purely pragmatic: To use it as a vehicle for demonstrating the usefulness of parametric procedural design; and as a non-trivial example domain that anybody, especially from “Old Europe”, can relate to by personal experience. Probably everybody has visited at some point a Gothic cathedral somewhere, and few individuals can deny that they have been impressed – whether they liked the style or not.

Artistic Impetus. Over time though, and through the confrontation with contemporal architects, a deeper subject arose. The often heard objection was that modern architecture can not be categorized using any rigid system of construction principles. Instead, modern architecture attempts to actually break rules, it intends to surprise, to be fresh and novel – at almost any price. Bluntly speaking, today’s architects consider themselves as artists, and architecture as an art discipline, rather than a skillful craft – which also explains the neglect of ‘architectural theory’ as a science. Buildings become pieces of art at the danger that their justification lies in the satisfaction of the architect’s subjective artistic impetus. The downside of this situation is that there is no longer a notion of ‘good’ or ‘bad’ architecture; buildings become incommensurable, each building is to be judged as an individual, much like paintings in a gallery. But the difference between art galleries and architecture is that people do have to live in their cities every day!

Discovering and expressing rules in architecture. Pursuing the project’s initial pragmatic pathway, a rather fascinating perspective for future work in the field of procedural modeling arises: To prove that also modern architecture does follow a set of rules, and that these rules are simple, maybe even very simple. How could this be done? To some degree there is a contradiction in that most modern architecture is planned using procedural CAD modeling software – but that the result claims to break all possible sets of rules! So this claim could in fact be proven wrong if a limited set of modeling operations could be identified, a tool-box that covers most of the patterns used in a variety of modern buildings. This would probably work best in a bottom-up fashion: By classifying stairways, windows, doors, typical room proportions, structural components, and so forth, and to do this for both ancient and modern styles. A comparison between the different resulting ‘tool boxes’, or stylistic vocabularies, might very well conclude that the classic, or ‘ancient’, styles are simply too complex for modern CAD systems. A very convincing example from many styles and era are the ubiquitous, abundant floral ornaments, carved into stone, which are repetitive but also slightly and skillfully varied with every repetition.

The potential of style libraries. A fascinating perspective of such endeavour is a partial exchange of rules and tools, such as to use the proportions of an ancient temple or a cathedral, but to apply today’s columns, portals, and steel arches. Vice versa, Gothic skyscrapers such as the Chrysler building in Manhattan come to mind, or even more distinct, the Chicago tribune tower in Chicago, which exhibits pointed arches in the top stories.

The second often heard argument is most prosaic: Stone masonry is simply too expensive, both in planning and in execution. But what if modern CAD tools knew about the construction principles, if they efficiently supported tools and typical operations for the manipulation of individual stones? The planning could be largely automatized, and execution could eventually be done by a CNC machine, or a robot with a chisel. And cost would no longer be much of an issue then. Would not popular demand require to have a look on which are the places and buildings in every city that are recognized and appreciated most by the visitors? Which places and squares people feel comfortable to be in, and where most people would agree: Here, in such an environment, I would like to live? An imaginary comparison of Venice or Florence to, e.g., any modern American or European suburb leads to the *architecture quality conjecture*: The quality of architecture *can* indeed be measured. What is maybe just lacking in this area is a systematic customer satisfaction evaluation. It would probably reveal some surprises to artistic architecture.

A new form of Architecture that respects the old proportions. So what remains to assemble is a sufficiently complete toolbox for, e.g., the Gothic style, one that permits to create typical buildings in all their complexity with the smallest possible amount of manual intervention. Yet despite its obvious procedural grounds and roots, there appears to be surprisingly few literature on the actual underlying geometric calculations! The most concrete and comprising exposition could only be found in a book a hundred years old: Georg Ungewitter’s *Lehrbuch der gotischen Konstruktionen* from 1858, especially in its delightful fourth edition from 1904. After 670 pages and more than 1500 figures, he comes to a very pointed conclusion that greatly summarizes what has been attempted to say in this final section.

It appears that today, after 150 years, his remarks still apply, and for the sake of its charming original style and historical authenticity, its German version shall conclude this thesis.

"Die schärfste Auffassung der zu erfüllenden Bedingungen, der gegebenen Verhältnisse und der Eigentümlichkeiten der Materialien, das Bestreben, immer die grössten Ziele mit den kleinsten Mitteln zu erreichen, vor allem aber die gewissenhafteste Scheu vor jeder Unwahrheit in der Formentwicklung und die dadurch bedingte gänzliche Vermeidung aller Surrogate sind die für die gotischen Konstruktionen charakteristischen Eigenschaften. Selbst die einer häufig vorkommenden Auffassung nach verderbtesten Werke der Spätgotik teilen dieselben, und sündigen nur durch eine gewisse Übertreibung, eine jedem Prinzip gefährliche Haarspalterei.

In nicht minderem Masse sind jene Eigenschaften auch die der griechischen Architektur, so dass die völlige Verschiedenheit der Resultate eben in der Verschiedenheit der Bedingungen und Materialien begründet ist, sowie ferner in der Zeitstellung und dem Entwicklungsgang der gotischen Kunst, wonach dieselbe in den Stand gesetzt war, auf den Resultaten aller vorangegangenen Kunstepochen, also auch jeder auf die griechische folgenden zu fussen und von denselben aus ihre Systeme zu entwickeln.

Hierin, in dem traditionellen Charakter der gotischen Kunst, in ihrer durchweg erhaltenen Geschichtlichkeit, liegt ein zweites nicht minder wichtiges Moment derselben, wodurch sie nicht so sehr von der Renaissance und dem Rokoko als von einer gewissen Richtung der modernen Kunstbestrebungen sich scheidet, welche dahin geht, die Erfindung eines neuen zeitgemässen Baustiles mittelst einer völlig willkürlichen Vermengung aller vorangegangenen auf dem Vehikel der Surrogate zu erjagen. Anstatt die Prinzipien der vorangegangenen Stile sich anzueignen, benascht man so ihre Resultate, anstatt die etwa der Neuzeit angehörigen Materialien, wie das Gusseisen, welche wirklich wertvolle Eigenschaften besitzen, den letzteren gemäss zu verwenden und eine entsprechende Formenentwicklung zu suchen, benutzt man sie vorherrschend als Täuschungsmittel zur Darlegung eines der ganzen Konstruktion fremden Reichtums, giesst sie in Formen, welchen ihren Eigenschaften völlig widersprechen, kurz man sucht eine freie künstlerische Thätigkeit dadurch zu erreichen, dass man alle Verstandesthätigkeit und selbst jedes tiefer gehende Studium völlig ausschliesst.

Um diese freie künstlerische Thätigkeit ist es nun überhaupt ein gar bedenkliches Ding. Mag es immerhin titanenhafte Individuen geben oder gegeben haben, vermögend von vornherein und mit einem Male die Elemente des früheren zu einem völlig neuen Ganzen zu verbinden, und so eine der Schöpfung fast adäquate künstlerische That zu thun, so ist doch der Glaube, in diese Kategorie zu gehören, für jeden Einzelnen sicher als ein Unglück anzusehen. Für alle nach minder grossartigem Massstab angelegte Naturen aber ist der einzige Weg zur künstlerischen Freiheit nur durch ein sorgfältiges Studium der vorangegangenen Kunstperioden zu finden, durch eine gewissenhafte Erforschung ihrer konstruktiven Prinzipien, mithin, da die gotische Architektur sich gewissermassen als der Abschluss und das Produkt aller primären Kunstperioden darstellt, zunächst in dem Studium dieser letzteren. Möchte es uns im Verlauf dieser Blätter gelungen sein, derartige Bestrebungen zu erleichtern."

Georg Ungewitter, "*Lehrbuch der gotischen Konstruktionen*", 1858, Nachwort

Bibliography

- [3DF94] Project to build a 3d fax machine. Marc Levoy and Brian Curless and Kari Pulli and others, 1994. graphics.stanford.edu/projects/faxing, also see the *Digital Michelangelo Project* at graphics.stanford.edu/projects/mich. 8
- [3DS] 3d studio max, animation, modeling, and rendering software. Discreet. at www.discreet.com. 255
- [AA96] AICHHOLZER O., AURENHAMMER F.: Straight skeletons for general polygonal figures in the plane. *Proc. 2nd Annu. Internat. Conf. Computing and Combinatorics* (1996), 117–126. 199
- [AA03] ADAMSON A., ALEXA M.: Approximating and intersecting surfaces from points. In *Proc. Symp. on Geometry Processing (SGP'03)* (Aachen, Germany, 2003), Eurographics/ACM SIGGRAPH, pp. 230–239. 20
- [AAAG95] AICHHOLZER O., AURENHAMMER F., ALBERTS D., GÄRTNER B.: A novel type of skeleton for polygons. *J.UCS: Journal of Universal Computer Science 1*, 12 (1995), 752–761. 199
- [ACK01a] AMENTA N., CHOI S., KOLLURI R.: The power crust. In *Proceedings of the 6th ACM Symposium on Solid Modeling and Applications* (Ann Arbor, Michigan, June 2001), pp. 249–260. 61
- [ACK01b] AMENTA N., CHOI S., KOLLURI R.: The power crust, unions of balls, and the medial axis transform. *Computational Geometry: Theory and Applications 19*, 2-3 (2001), 127–153. (special issue on surface reconstruction). 61
- [ACS02] AKLEMAN E., CHEN J., SRINIVASAN V.: A prototype system for robust, interactive and user-friendly modeling of orientable 2-manifold meshes. In *SMI '02: Proceedings of the Shape Modeling International 2002 (SMI'02)* (Washington, DC, USA, 2002), IEEE Computer Society, p. 43. 27
- [AD03] ADAMS B., DUTRÉ P.: Interactive boolean operations on surfel-bounded solids. *ACM Transactions on Graphics 22*, 3 (2003), 651–656. 25
- [ADG*03] ALEXA M., DACHSBACHER C., GROSS M., PAULY M., VAN BAAR J., ZWICKER M.: Point-based computer graphics. In *Eurographics 2003 Conference Tutorial T1*. Eurographics Association, 2003. 19
- [Ado99] ADOBE SYSTEMS INC.: *PostScript Language Reference Manual*, 3 ed. Addison-Wesley, 1999. 50, 215, 218, 220
- [AFM*00] ANDERSON D., FRANKEL J. L., MARKS J., AGARWALA A., BEARDSLEY P., HODGINS J., LEIGH D., RYALL K., SULLIVAN E., YEDIDIA J. S.: Tangible interaction + graphical interpretation: a new approach to 3d modeling. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2000), ACM Press/Addison-Wesley Publishing Co., pp. 393–402. 26
- [AFRS03] ATTENE M., FALCIDIENO B., ROSSIGNAC J., SPAGNUOLO M.: Edge-sharpener: recovering sharp features in triangulations of non-adaptively re-meshed surfaces. In *Proc. Symp. on Geometry Processing (SGP'03)* (Aachen, Germany, 2003), Eurographics/ACM SIGGRAPH, pp. 62–69. 20
- [Aim04] AIM@SHAPE: Project website of fp6 ist noe 506766, 2004. www.aim-at-shape.net. 29
- [App04] APPLE COMPUTER, INC.: *Apple Human Interface Guidelines*, 2004-05-27 ed. Cupertino, CA 95014, USA, 2004. developer.apple.com/documentation. 265
- [B*] BEAZLEY D., ET AL.: Swig wrapper generator website. www.swig.org. 36

- [Baj96] BAJAJ C.: Free-form modeling with implicit surface patches. In *Implicit Surfaces*, Bloomenthal J., Wyvill B., (Eds.). Morgan Kaufman Publishers, 1996. 161
- [Bau75] BAUMGART B. G.: A polyhedron representation for computer vision. In *Nat. Comp. Conf. 44* (1975), AFIPS, pp. 589–596. 151
- [BBB*97] BLOOMENTHAL J., BAJAJ C., BLINN J., CANI-GASCUEL M.-P., ROCKWOOD A., WYVILL B., , WYVILL G.: *Introduction to Implicit Surfaces*. Morgan Kaufman, 1997. 19
- [BBC*99] BAJAJ C. L., BALDAZZI C., CUTCHIN S., PAOLUZZI A., PASCUCCI V., VINCENTINO M.: A programming approach for complex animations. *Computer Aided Design* 31, 11 (1999). 28
- [BCCD04] BOURGUIGNON D., CHAINE R., CANI M.-P., DRETTAKIS G.: Relief: A modeling by drawing tool. In *First Eurographics Workshop on Sketch-Based Interfaces and Modeling (SBM'04)* (Grenoble, France, sep 2004), Hughes J., Jorge J., (Eds.), Eurographics, pp. 151–160. 24
- [BCF*04] BALZANI M., CALLIERI M., FABBRI M., FASANO A., MONTANI C., PINGI P., SANTOPUOLI N., SCOPIGNO R., UCCELLI F., VARONE A.: Digital representation and multimodal presentation of archeological graffiti at pompeii. In *Proceedings of VAST 2004* (Brussels, Belgium, Dec. 2004), et al. K. C., (Ed.), Eurographics Association, pp. 93–104. 8
- [BCX95] BAJAJ C. L., CHEN J., XU G.: Modeling with cubic A-patches. *ACM Transactions on Graphics* 14, 2 (Apr. 1995), 103–133. 161
- [Ber96] BERNHARD F. (Ed.): *Der Steinmetz und Steinbildhauer: Ausbildung und Praxis*. Verlag Callwey, München, Germany, 1996. 229
- [BFH95] BENDELS H., FELLNER D. W., HAVEMANN S.: Modellierung der Grundlagen — Erweiterbare Datenstrukturen zur Modellierung und Visualisierung polygonaler Welten. In *Proc. Intern. Workshop Modeling – Virtual Worlds – Distributed Graphics*, Fellner D. W., (Ed.). infix, 1995, pp. 149–158. 39
- [BFH*98] BUHMANN J. M., FELLNER D. W., HELD M., KETTERER J., PUZICHA J.: Dithered color quantization. *Computer Graphics Forum* 17, 3 (1998), 219–232. 177
- [BFH05] BERNDT R., FELLNER D. W., HAVEMANN S.: Generative 3d models: A key to more information within less bandwidth at higher quality. In *Web3D '05: Proceedings of the tenth international conference on 3D Web technology* (New York, NY, USA, 2005), ACM Press, pp. 111–121. 49
- [BGK03] BALÁZS A., GUTHE M., KLEIN R.: *Fat borders: Gap filling for effective view-dependent lod rendering*. Tech. rep., Universität Bonn, June 2003. 161
- [Bin89] BINDING G.: *Maßwerk*. Wissenschaftliche Buchgesellschaft, Darmstadt, 1989. ISBN: 3-534-01582-7. 229, 234
- [Bin02] BINDING G.: *Hochgotik*. Taschen Verlag, Cologne, Germany, 2002. 229, 234
- [BK04] BOTSCH M., KOBELT L.: An intuitive framework for real-time freeform modeling. *ACM Trans. Graph.* 23, 3 (2004), 630–634. 23
- [Bli82] BLINN J. F.: A generalization of algebraic surface drawing. *ACM Trans. Graph.* 1, 3 (1982), 235–256. 22
- [BLZ00] BIERMANN H., LEVIN A., ZORIN D.: Piecewise smooth subdivision surfaces with normal control. In *Proceedings of SIGGRAPH 2000* (2000), pp. 113–120. 103, 104
- [BM03] BOIER-MARTIN I. M.: Domain decomposition for multiresolution analysis. In *Proc. Symp. on Geometry Processing (SGP'03)* (Aachen, Germany, 2003), Eurographics/ACM SIGGRAPH, pp. 31–40. 20
- [BMBZ02] BIERMANN H., MARTIN I., BERNARDINI F., ZORIN D.: Cut-and-paste editing of multiresolution surfaces. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2002), ACM Press, pp. 312–321. 23
- [BO03] BOISSONNAT J. D., OUDOT S.: Provably good surface sampling and approximation. In *Proc. Symp. on Geometry Processing (SGP'03)* (Aachen, Germany, 2003), Eurographics/ACM SIGGRAPH, pp. 9–18. 20

- [Boint] BOISSERÉE S.: *Ansichten, Risse und einzelne Teile des Domes von Köln*. Arnold Wolff, Verlag Kölner Dom, Köln, 1979 (reprint). 250
- [Bot] BOTSCH M.: Openmesh documentation. www.openmesh.org. 133
- [Bri04] BRICKLIN D.: Visicalc: Information from its creators. In *Bricklin.com*. WWW, 2004. 57
- [Bro82] BROWN C. M.: Padl-2: a technical summary. *IEEE Computer Graphics & Applications* 2 (Mar. 1982), 69–84. 12
- [BS02a] BOLZ J., SCHRÖDER P.: Rapid evaluation of catmull-clark subdivision surfaces. In *Proc. Web3D 2002 Symposium* (2002). 161
- [BS02b] BOLZ J., SCHROEDER P.: Rapid evaluation of catmull-clark subdivision surfaces. In *Proceedings of Web3D'02* (2002), pp. 11–17. 118, 126
- [BSB*01] BROWN J., SORKIN S., BRUYNS C., LATOMBE J., MONTGOMERY K., STEPHANIDES M.: Real-time simulation of deformable objects: Tools and application. In *Computer Animation 2001* (Seoul, Korea, Nov 6-8 2001). 23
- [BSBI02] BOTSCH M., STEINBERG S., BISCHOFF S., LT L. K.: Openmesh - a generic and efficient polygon mesh data structure. In *Proceedings of OpenSG Symposium 2002* (2002). 133
- [Cam98] CAMPAGNA S.: *Polygonreduktion*. PhD thesis, Universität Erlangen-Nürnberg, 1998. 142
- [Cat] Catia digital product definition and simulation software. Dassault Systemes. at www.3ds.com. 255
- [CBC*01] CARR J. C., BEATSON R. K., CHERRIE J. B., MITCHELL T. J., FRIGHT W. R., MCCALLUM B. C., EVANS T. R.: Reconstruction and representation of 3d objects with radial basis functions. In *Proceedings of ACM SIGGRAPH 2001* (Aug. 2001), Computer Graphics Proceedings, Annual Conference Series, pp. 67–76. 19
- [CC78] CATMULL E., CLARK J.: Recursively generated b-spline surfaces on arbitrary topological mesh es. *Computer Aided Design* 10, 6 (1978), 350–355. 91
- [CCG*04] CALLIERI M., CIGNONI P., GANOVELLI F., IMPOCO G., MONTANI C., PINGI P., PONCHIO F., SCOPIGNO R.: Visualization and 3d data processing in david's restoration. *IEEE Computer Graphics & Applications* 24, 2 (March/April 2004), 16–21. 8
- [CCH96] CAPOYLEAS V., CHEN X., HOFFMANN C.: Generic naming in generative constraintbased design. *Computer-Aided Design* 28, 1 (1996), 17–26. 13
- [CDES01] CHENG H.-L., DEY T. K., EDELSBRUNNER H., SULLIVAN J.: Dynamic skin triangulation. *Discrete Comput. Geom.* 25, 4 (2001), 525–568. 19, 22, 162
- [CDM*02] CUTLER B., DORSEY J., MCMILLAN L., MÜLLER M., JAGNOW R.: A procedural approach to authoring solid models. *ACM Transactions on Graphics* 21, 3 (July 2002), 302–311. 27, 61
- [CFM*94] CIGNONI P., FLORIANI L. D., MONTANI C., PUPPO E., SCOPIGNO R.: Multiresolution modeling and visualization of volume data based on simplicial complexes. In *ACM Symposium on Volume Visualization* (Washington, Oct 1994), pp. 19–26. 161
- [CGC*02] CAPELL S., GREEN S., CURLESS B., DUCHAMP T., POPOVIC Z.: A multiresolution framework for dynamic deformations. In *Proc. ACM SIGGRAPH Symposium on Computer Animation* (2002). 23
- [Cha74] CHAIKIN G.: An algorithm for high-speed curve generation. *Computer Graphics and Image Processing* 3 (1974), 346–349. 91
- [Cha03] CHASE S. C.: Revisiting the use of generative design tools in the early stages of design education. In *Digital Design, Proc. 21st Conference on Education in Computer Aided Architectural Design in Europe (eCAADe)* (Graz, 2003), Dokonal W., Hirschberg U., (Eds.), pp. 465–472. 27
- [CKS98] CAMPAGNA S., KOBBELT L., SEIDEL H.-P.: Directed edges — A scalable representation for triangle meshes. *Journal of Graphics Tools: JGT* 3, 4 (1998), 1–12. 142

- [CLR90] CORMEN T. H., LEISERSON C. E., RIVEST R. L.: *Introduction to Algorithms*. MIT Press, 1990. 136, 184
- [CN00] CHUA C., NEUMANN U.: Hardware-accelerated free-form deformation. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (New York, NY, USA, 2000), ACM Press, pp. 33–39. 22
- [Coe88] COENEN U.: *Die spätgotischen Werkmeisterbücher in Deutschland als Beitrag zur mittelalterlichen Architekturtheorie – Untersuchung und Edition der Lehrschriften für Entwurf und Ausführung von Sakralbauten*. PhD thesis, Technische Hochschule RWTH Aachen, Aachen, 1988. 229
- [CRE01] COHEN E., RIESENFELD R., ELBER G.: *Geometric Modeling with Splines*. AK Peters, 2001. 91
- [CSW95] CHIN F., SNOEYINK J., WANG C.-A.: Finding the medial axis of a simple polygon in linear time. In *Proc. 6th Annu. Internat. Sympos. Algorithms Comput.* (1995), Lecture Notes Comput. Sci. 1004, Springer-Verlag, pp. 382–391. 199
- [CV02] CHENG S.-W., VIGNERON A.: Motorcycle graphs and straight skeletons. In *Proc. 13th ACM/SIAM Symp. on Discrete Algorithms* (2002), pp. 156–165. 199
- [DAHFO4] DAY A. M., ARNOLD D. B., HAVEMANN S., FELLNER D. W.: Combining polygonal and subdivision surface approaches to modelling and rendering of urban environments. *Computers & Graphics* 28, 4 (Aug. 2004), 497–507. (also in *Proc. International Conference on Cyberworlds 2003*, Singapore, Dec 2003). 44
- [DDCB01] DEBUNNE G., DESBRUN M., CANI M.-P., BARR A. H.: Dynamic real-time deformations using space and time adaptive sampling. In *Proc. SIGGRAPH 2001* (2001), ACM Press. 23, 161
- [Dev] 3d engines database. DevMaster.net. www.devmaster.net/engines. 17
- [DHQed] DUAN Y., HUA J., QIN H.: Interactive shape modeling using lagrangian surface flow. *The Visual Computer* (2005 (accepted)). 22
- [Dir] Directx. Microsoft DirectX SDK Documentation. www.microsoft.com/directx. 17
- [DS78] DOO D., SABIN M.: Behaviour of recursive division surfaces near extraordinary points. *Computer Aided Design* 10, 6 (1978), 356–360. 91
- [DTG96] DESBRUN M., TSINGOS N., GASCUEL M.-P.: Adaptive sampling of implicit surfaces for interactive modelling and animation. *Computer Graphics Forum* 15, 5 (Dec. 1996), 319–325. 161
- [dvOS97] DE BERG M., VAN KREVELD M., OVERMARS M., SCHWARZKOPF O.: *Computational Geometry – Algorithms and Applications*. Springer, 1997. 157, 187
- [eCA] ECAADE W.: Education and research in computer aided architectural design in europe (non-profit organization founded in 1983). www.ecaade.org. 27
- [Ede99] EDELSBRUNNER H.: Deformable smooth surface design. *Discrete Comput. Geom.*, 21 (1999), 87–115. 22
- [EE99] EPPSTEIN D., ERICKSON J.: Raising roofs, crashing cycles, and playing pool: Applications of a data structure for finding pairwise interactions. *Disc. & Comput. Geometry* 22 (1999), 569–592. 199
- [EF05] EGLE J., FIECHTER E. R.: *Baustil- und Bauformenlehre in Abbildungen mit Texterläuterungen*, vol. 3, *Gotische Baukunst*. Stuttgart, Hannover, 1905. reprint from 1996, Edition Libri Rari, Verlag Th. Schäfer, Hannover, Germany. 229, 232, 234, 236, 247, 250
- [EF00] ENDRES A., FELLNER D.: *Digitale Bibliotheken, Informatik-Lösungen für globale Wissensmärkte*. Die Deutsche Bibliothek, dpunkt.verlag, Heidelberg, 2000. 34
- [EH97] ELLIOTT C., HUDAK P.: Functional reactive animation. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 1997), ACM Press, pp. 263–273. 28
- [EKE01] ENGEL K., KRAUS M., ERTL T.: High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (New York, NY, USA, 2001), ACM Press, pp. 9–16. 19

- [Eil96] ELLIOTT C.: *A Brief Introduction to ActiveVRML*. Tech. Rep. MSR-TR-96-05, Microsoft Research, 1996. conal.net. 28
- [Eil04] ELLIOTT C.: Programming graphics processors functionally. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell* (New York, NY, USA, 2004), ACM Press, pp. 45–56. 28
- [Els] The electronic tax declaration. Federal Ministry of Finance. www.elster.de. 267
- [Epp92] EPPSTEIN D.: Approximating the minimum weight triangulation. In *SODA '92: Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms* (Philadelphia, PA, USA, 1992), Society for Industrial and Applied Mathematics, pp. 48–57. 158
- [ES90] ELLIS M. A., STROUSTRUP B.: *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Massachusetts, Dec. 1990. 134, 137, 139, 140
- [ESYAE95] ELLIOTT C., SCHECHTER G., YEUNG R., ABI-EZZI S.: TBAG: A high level framework for interactive, animated 3D graphics applications. In *Proc. SIGGRAPH '94* (1995), ACM, pp. 421–434. 28
- [Far02] FARIN G.: *Curves and Surfaces for CAGD*, 5th ed. Morgan Kaufmann Publishers, 2002. 18, 35, 89, 97, 161
- [FCG99] FERLEY E., CANI M.-P., GASCUEL J.-D.: Practical volumetric sculpting. In *Proceedings of Implicit Surface '99* (Sep 1999). 21
- [Fel92] FELLNER D. W.: *Computer Grafik*, 2 ed., vol. 58 of *Reihe Informatik*. B.I. Wissenschaftsverlag, Mannheim, 1992. 39, 177
- [FF97] FIELL C., FIELL P.: *1000 chairs*. Taschen Verlag, Cologne, Germany, 1997. 2
- [FFCg] FARBRAUSCH, FIVER2, CHAOS, GIZMO: Theprodukt and farbrausch demoscene project website, providing the werkzeug tool and kkrieger. www.opensg.org. 255
- [FGK*00] FABRI A., GIEZEMAN G.-J., KETTNER L., SCHIRRA S., SCHÖNHERR S.: On the design of cgal, a computational geometry algorithms library. *Softw. – Pract. Exp.* 30, 11 (2000), 1167–1202. Special Issue on Discrete Algorithm Engineering. 204
- [FH02] FELLNER D. W., HAVEMANN S.: Digital libraries. In *Proc. EWV02 – East-West-Vision 2002* (Graz, Austria, sep 12-13 2002), Leberl F., Teller S., (Eds.), OCG, pp. 213–221. 34
- [FH03] FLORIANI L. D., HUI A.: A scalable data structure for three-dimensional non-manifold objects. In *Proc. Symp. on Geometry Processing (SGP'03)* (Aachen, Germany, 2003), Eurographics/ACM SIGGRAPH, pp. 72–82. 20
- [FHH*00] FELLNER D. W., HABER J., HAVEMANN S., KOBBELT L., LENSCH H. P. A., MÜLLER G., PETER I., SCHNEIDER R., SEIDEL H.-P., STRASSER W.: Beiträge der Computergraphik zur Realisierung eines verallgemeinerten Dokumentenbegriffs. *it+ti Informationstechnik und Technische Informatik* 42, 6 (2000), 8–16. 34
- [FHH03] FELLNER D. W., HAVEMANN S., HOPP A.: Dave – eine neue technologie zur preiswerten und hochqualitativen immersiven 3d-darstellung. In *Proc. 8. Workshop: Sichtsysteme – Visualisierung in der Simulationstechnik* (Bremen, nov 2003), Möller R., (Ed.), Shaker Verlag, pp. 77–87. 45
- [FHM98] FELLNER D. W., HAVEMANN S., MÜLLER G.: Modeling of and navigation in complex 3d documents. *Computers & Graphics* 22, 6 (Dec. 1998), 647–653. 34
- [FKS*04] FUNKHOUSER T., KAZHDAN M., SHILANE P., MIN P., KIEFER W., TAL A., RUSINKIEWICZ S., DOBKIN D.: Modeling by example. *ACM Trans. Graph.* 23, 3 (2004), 652–663. 25
- [Fla03] FLANAGAN R. H.: Generative logic in digital design. In *Digital Design, Proc. 21st Conference on Education in Computer Aided Architectural Design in Europe (eCAADe)* (Graz, 2003), Dokonal W., Hirschberg U., (Eds.), pp. 473–483. 27
- [FS97] FELLNER D. W., SCHENK N.: MRT – a tool for simulations in 3D geometric domains. In *11th European Simulation Multiconference*, Kaylan A. R., Lehmann A., (Eds.). Society for Computer Simulation Intl. (SCS), 1997, pp. 185–188. 39

- [Gai00] GAIN J.: *Enhancing Spatial Deformation for Virtual Sculpting*. PhD thesis, The Computer Laboratory, University of Cambridge, June 2000. Technical Report TR499. 161, 162
- [GD99] GAIN J., DODGSON N.: Adaptive refinement and decimation under free-form deformation. In *Eurographics UK '99* (Cambridge(UK), April 13-15 1999). 161, 162
- [GGH02] GU X., GORTLER S. J., HOPPE H.: Geometry images. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2002), ACM Press, pp. 355–361. 23
- [GH97] GARLAND M., HECKBERT P. S.: Surface simplification using quadric error metrics. In *Proceedings of SIGGRAPH 97* (August 1997), ACM SIGGRAPH, pp. 209–216. 144
- [Gib84] GIBSON W.: *Neuromancer*. Ace Books, New York, 1984. 17
- [Gla04] GLASSNER A.: Andrew glassner's notebook: Crop art. *IEEE Computer Graphics & Applications* 24, 5 (sep/oct 2004), 86–99. 51
- [GP95] GRIMM C., PUGMIRE D.: Visual interfaces for solids modeling. In *UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology* (New York, NY, USA, 1995), ACM Press, pp. 51–60. 25, 37
- [GS85] GUIBAS L., STOLFI J.: Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics* 4, 2 (Apr. 1985), 74–123. 151
- [GVSS00] GUSKOV I., VIDIMCE K., SWELDENS W., SCHRÖDER P.: Normal meshes. *Proceedings of SIGGRAPH 2000* (July 2000), 95–102. ISBN 1-58113-208-5. 23
- [GY03] GU X., YAU S.-T.: Global conformal surface parameterization. In *Proc. Symp. on Geometry Processing (SGP'03)* (Aachen, Germany, 2003), Eurographics/ACM SIGGRAPH, pp. 127–137. 20
- [Hal] Halfife (ego shooter game and game engine). Valve Software Inc. www.valvesoftware.com. 262
- [Hav97] HAVEMANN S.: *Generative Modellierung*. Master's thesis, Universität Bonn, Deutschland, dec 1997. Diplomarbeit. 33
- [Hav99] HAVEMANN S.: Effizienter austausch von 3d dokumenten auf basis von generativer modellierung (invited). In *GI Jahrestagung Informatik '99 - Informatik überwindet Grenzen* (Paderborn, okt 1999), Beiersdörfer K., Engels G., Schäfer W., (Eds.), Springer, pp. 164–172. 34
- [Hav02a] HAVEMANN S.: Höhlenzeitalter – cave-instaiiation auf linux-pcs. *iX Magazin für professionelle Informationstechnik*, 11 (nov 2002), 99–103. 45
- [Hav02b] HAVEMANN S.: Interactive rendering of catmull/clark surfaces with crease edges. *The Visual Computer* 18, 5/6 (2002), 286–298. 41, 119, 127, 128, 129
- [HBJF02] HART J. C., BACHTA E., JAROSZ W., FLEURY T.: Using particles to sample and control more complex implicit surfaces. In *Proc. Shape Modeling International (SMI 2002)* (Banff, Canada, May 2002), IEEE Computer Society, pp. 129–136. 22
- [HDD*94] HOPPE H., DEROSE T., DUCHAMP T., HALSTEAD M., JIN H., McDONALD J., SCHWEITZER J., STUETZLE W.: Piecewise smooth surface reconstruction. *Proc. SIGGRAPH 94* (July 1994), 295–302. 175
- [Hei94] HEISSERMAN J.: Generative geometric design. *IEEE Comput. Graph. Appl.* 14, 2 (1994), 37–45. 27
- [HF01] HAVEMANN S., FELLNER D.: A versatile 3d model representation for cultural reconstruction. In *Proc. VAST '01: Proceedings of the 2001 conference on Virtual reality, archeology, and cultural heritage* (New York, NY, USA, 2001), ACM Press, pp. 205–212. 44, 162
- [HF04a] HAVEMANN S., FELLNER D.: Generative parametric design of gothic window tracery (short paper). In *Proc. Shape Modeling and Application (SMI'04)* (Genova, June 2004), Giannini F., Pasko A., (Eds.), IEEE, pp. 350–354. 55, 232

- [HF04b] HAVEMANN S., FELLNER D. W.: Generative parametric design of gothic window tracery. In *Proc. VAST 2004: The 5th International Symposium on Virtual Reality, Archaeology and Cultural Heritage* (Brussels, Belgium, 2004), Cain K., Chrysanthou Y., Niccolucci F., Silberman N., (Eds.), Eurographics, pp. 193–201. [55](#), [232](#)
- [HF05a] HAVEMANN, FELLNER: Progressive combined breps – multi-resolution meshes for incremental real-time shape manipulation (submitted for publication). [41](#)
- [HF05b] HAVEMANN S., FELLNER D.: Managing procedural knowledge. In *Proc. I-Know 05* (Graz, Austria, July 2005). (to be published). [58](#)
- [HFDA03] HAVEMANN S., FELLNER D. W., DAY A., ARNOLD D. B.: New approaches to efficient rendering of complex reconstructed environments. In *VAST 2003: The 4th International Symposium on Virtual Reality, Archaeology and Cultural Heritage* (Brighton, UK, Nov 2003), Eurographics, pp. 185–193. [44](#)
- [HJA02] HOFFMANN C., JOAN-ARINYO R.: Parametric modeling. In *Handbook of Computer Aided Geometric Design*. Elsevier, 2002, ch. 21, pp. 519–541. [12](#), [13](#), [256](#)
- [HKD93] HALSTEAD H., KASS M., DEROSE T.: Efficient, fair interpolation using catmull–clark surfaces. In *Proceedings of COMPUTER GRAPHICS 1993* (1993), pp. 35–44. [100](#)
- [HL92] HOSCHEK J., LASSER D.: *Grundlagen der geometrischen Datenverarbeitung*, 2 ed. Teubner, Stuttgart, 1992. [35](#), [89](#)
- [Hop96] HOPPE H.: Progressive meshes. In *Proceedings of SIGGRAPH 96* (New Orleans, Louisiana, August 1996), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 99–108. [146](#)
- [Hop97] HOPPE H.: View-dependent refinement of progressive meshes. *Proceedings of SIGGRAPH 97* (August 1997), 189–198. ISBN 0-89791-896-7. Held in Los Angeles, California. [146](#), [150](#), [187](#)
- [Hop98] HOPPE H.: Efficient implementation of progressive meshes. *Computers & Graphics* 22, 1 (1998), 27–36. [144](#)
- [Hop99] HOPPE H.: Optimization of mesh locality for transparent vertex caching. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1999), ACM Press/Addison-Wesley Publishing Co., pp. 269–276. [158](#)
- [Hou] Houdini animation, modeling, and rendering software. Side Effects Software. at www.sidefx.com. [255](#)
- [HPS04] HANSMANN W., PURGATHOFER W., SILLION F. (Eds.): *Proc. Eurographics Workshop on Sketch-Based Interfaces and Modeling* (Grenoble, France, August 2004), Eurographics Association. available from diglib.org/EG/DL/WS/SBM/SBM04. [24](#)
- [HQ01] HUA J., QIN H.: Haptic sculpting of volumetric implicit functions. In *Proc. Pacific Graphics 2001* (Tokyo, Japan, Oct 2001), pp. 254–264. [161](#)
- [HW93] HEISSERMAN J., WOODBURY R.: Generating languages of solid models. In *SMA '93: Proceedings of the Second Symposium on Solid Modeling and Applications* (May 1993), pp. 103–112. [27](#)
- [IG03] ISENBURG M., GUMHOLD S.: Out-of-core compression for gigantic polygon meshes. *ACM Transactions on Graphics* 22, 3 (July 2003), 935–942. [8](#), [146](#), [228](#)
- [IGE96] Iges 5.3, initial graphics exchange specification, ans us pro/ipo-100-1996. American National Standard, 1996. available from www.nist.gov/iges. [13](#)
- [IMT99] IGARASHI T., MATSUOKA S., TANAKA H.: Teddy: A sketching interface for 3d freeform design. In *Proceedings of SIGGRAPH 99* (Aug. 1999), Computer Graphics Proceedings, Annual Conference Series, pp. 409–416. [24](#)
- [Int98] INTEL CORPORATION: *Intel Architecture Optimization Reference Manual*, 1998. Order Number: 730795-001. [126](#)

- [Int99a] INTEL CORPORATION: *IA-32 Intel Architecture Software Developer's Manual*, 1999. Order Number: 245471-010. 118, 124
- [Int99b] INTEL CORPORATION: *Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual*, 1999. Order Number: 248966-007. 124
- [jpe00] Jpeg 2000 standard. ISO/IEC, 2000. ISO/IEC 15444-1:2000, available from www.jpeg.org/jpeg2000, also see www.iso.org. 20, 31, 146
- [JTO04] Jt open – an open platform for visualization, collaboration and data sharing across the product lifecycle. UGS - The PLM Company, 2004. available from www.jtopen.com. 16
- [KAHF05] KIM H., ALBUQUERQUE G., HAVEMANN S., FELLNER D.: Tangible 3d: Immersive 3d modeling through hand gesture interaction. In *Proc. 11. Eurographics Workshop on Environments IPT&EGVE 05* (Aalborg, Oct 2005), Eurographics/ACM Siggraph, ACM press (planned). 56
- [KAP*03] KARTASHEVA E., ADZHIEV V., PASKO A., FRYAZINOV O., GASILOV V.: Discretization of functionally based heterogeneous objects. In *SM '03: Proceedings of the eighth ACM symposium on Solid modeling and applications* (New York, NY, USA, 2003), ACM Press, pp. 145–156. 28
- [KB05] KOBBELT L., BOTSCH M.: A survey of point-based techniques in computer graphics. *to appear in Computers & Graphics* (2005). 22
- [KBB*00] KOBBELT L., BISCHOFF S., BOTSCH M., KÄHLER K., RÖSSL C., SCHNEIDER R., VORSATZ J.: Geometric modeling based on polygonal meshes. In *Eurographics 2000 Tutorial*. Eurographics Association, 2000. 162
- [KBJ04] KAWAHARA H., BYRNE P., JOHNSON D.: *Project Looking Glass API Design Overview*, Oct. 2004. <https://lg3d-core.dev.java.net>. 265
- [KBS00] KOBBELT L., BAREUTHER T., SEIDEL H.-P.: Multiresolution shape deformations for meshes with dynamic vertex connectivity. *Computer Graphics Forum 19* (2000), C249–C260. Proc. Eurographics 2000. 23, 162
- [KCVS98] KOBBELT L., CAMPAGNA S., VORSATZ J., SEIDEL H.-P.: Interactive multi-resolution modeling on arbitrary meshes. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1998), ACM Press, pp. 105–114. 23
- [Ket99] KETTNER L.: Using generic programming for designing a data structure for polyhedral surfaces. *Computational Geometry 13*, 1 (1999), 65–90. 151
- [KHR02] KARPENKO O., HUGHES J. F., RASKAR R.: Free-form sketching with variational implicit surfaces. *Computer Graphics Forum* (Sep. 2002). 24
- [Kin93] KINSEY L. C.: *Topology of Surfaces*. Springer-Verlag, 1993. 59, 61, 66, 68, 77
- [KML96] KUMAR S., MANOCHA D., LASTRA A.: Interactive display of large nurbs models. *IEEE Transactions on Visualization and Computer Graphics 2*, 4 (December 1996). 161
- [KN04] KETTNER L., NÄHER S.: Two computational geometry libraries: Leda and cgal. In *Handbook of Discrete and Computational Geometry*, Goodman J. E., O'Rourke J., (Eds.), second ed. CRC Press LLC, Boca Raton, FL, 2004, ch. 64, pp. 1435–1463. 204
- [Koc00] KOCH W.: *Baustilkunde : das Standardwerk zur europäischen Baukunst von der Antike bis zur Gegenwart*, 22 ed. Bertelsmann-Lexikon-Verlag, Gütersloh, 2000. ISBN: 3-577-10480-5. 229
- [Kre00] KREBS B.: *Probabilistische Erkennung von 3d-Freifformobjekten mit Bayesschen Netzen*. PhD thesis, Technische Universität Braunschweig, 2000. 20
- [KV03] KALAI AH A., VARSHNEY A.: Statistical point geometry. In *Proc. Symp. on Geometry Processing (SGP'03)* (Aachen, Germany, 2003), Eurographics/ACM SIGGRAPH, pp. 107–115. 20
- [Lag04] LAGEMANN M.: *Verkleben zweier BRep Flächen mit Hilfe von Euler Operatoren*. Tech. rep., Technische Universität Braunschweig, 2004. Studienarbeit Wintersemester 2003/2004. 42

- [LCZ99] LIEBOWITZ D., CRIMINISI A., ZISSERMAN A.: Creating architectural models from images. In *Proc. EuroGraphics* (Sept. 1999), vol. 18, pp. 39–50. 8
- [LE97] LUEBKE D., ERIKSON C.: View-dependent simplification of arbitrary polygonal environments. *Proc. SIGGRAPH 97* (August 1997), 199–208. 175, 187
- [Leg] Lego homepage. The LEGO Group. www.lego.com. 2
- [Ley01] LEYTON M.: *A Generative Theory of Shape*. Lecture Notes in Computer Science, Volume 2145. Springer-Verlag, 2001. 29
- [LF03] LAWRENCE J., FUNKHOUSER T.: A painting interface for interactive surface deformations. In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications* (Washington, DC, USA, 2003), IEEE Computer Society, p. 141. 24
- [LH03] LAWLOR O. S., HART J. C.: Bounding recursive procedural models using convex optimization. In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications* (Washington, DC, USA, 2003), IEEE Computer Society, p. 283. 30
- [LH04] LACZ P., HART J. C.: Procedural geometric synthesis on the gpu. In *Manuscript accompanying poster at GP²: The ACM Workshop on General Purpose Computing on Graphics Processors* (also as SIGGRAPH 2004 poster, Aug 2004). available from graphics.cs.uiuc.edu/~jch. 25
- [LHSW03] LOSASSO F., HOPPE H., SCHAEFER S., WARREN J.: Smooth geometry images. In *Proc. Symp. on Geometry Processing (SGP'03)* (Aachen, Germany, 2003), Eurographics/ACM SIGGRAPH, pp. 138–145. 20
- [Lie03] LIEPA P.: Filling holes in meshes. In *Proc. Symp. on Geometry Processing (SGP'03)* (Aachen, Germany, 2003), Eurographics/ACM SIGGRAPH, pp. 200–205. 20
- [LKG*03] LLAMAS I., KIM B., GARGUS J., ROSSIGNAC J., SHAW C. D.: Twister: a space-warp operator for the two-handed editing of 3d shapes. *ACM Trans. Graph.* 22, 3 (2003), 663–668. 22
- [LL99] LI F., LAU R.: Real-time rendering of deformable parametric free-form surfaces. In *Proc. ACM Symposium on VR Software and Technology (VRST)* (Dec 1999), pp. 131–138. 161
- [LLS01] LITKE N., LEVIN A., SCHRÖDER P.: Fitting subdivision surfaces. In *VIS '01: Proceedings of the conference on Visualization '01* (Washington, DC, USA, 2001), IEEE Computer Society, pp. 319–324. 23
- [LPC*00] LEVOY M., PULLI K., CURLESS B., RUSINKIEWICZ S., KOLLER D., PEREIRA L., GINZTON M., ANDERSON S., DAVIS J., GINSBERG J., SHADE J., FULK D.: The digital michelangelo project: 3d scanning of large statues. In *Proceedings of ACM SIGGRAPH 2000* (July 2000), Computer Graphics Proceedings, Annual Conference Series, pp. 131–144. 146
- [Män88] MÄNTYLÄ M.: *An Introduction to Solid Modeling*. Computer Science Press, Rockville, 1988. 41, 59, 60, 78, 86, 209
- [Map] Maple 9.5 computer algebra system. Maplesoft, Waterloo Maple Inc. www.maplesoft.com. 118
- [May] Maya integrated 3d modeling, animation, effects, and rendering solution. Alias Systems Corporation. 255
- [MBH*01] MEISSNER M., BARTZ D., HÜTTNER T., MÜLLER G., EINIGHAMMER J.: Generation of decomposition hierarchies for efficient occlusion culling of large polygonal models. In *Proc. Vision, Modeling, and Visualization (VMV 2001)* (Stuttgart, Nov 2001). also available as technical report TUBSCG-1999-6-1 from the Institute of ComputerGraphics, TU Braunschweig. 34
- [MBWB02] MUSETH K., BREEN D. E., WHITAKER R. T., BARR A. H.: Level set surface editing operators. *ACM Transactions on Graphics* 21, 3 (July 2002), 330–338. 21
- [MCCH99] MARKOSIAN L., COHEN J. M., CRULLI T., HUGHES J.: Skin: a constructive approach to modeling free-form shapes. *Computer Graphics* 33, Annual Conference Series (1999), 393–400. 22, 161
- [McD03] McDONNELL K. T.: *DYNASOAR: DYNAMIC Solid Objects of ARbitrary topology*. PhD thesis, Stony Brook University, August 2003. (Advisor: Professor Hong Qin). 23, 161

- [Meh84] MEHLHORN K.: *Data Structures and Algorithms*, vol. 3: Multi-dimensional Searching and Computational Geometry. Springer Verlag, 1984, ch. VIII.4.1 and 4.2, pp. 147–172. 157
- [MH00] MÜLLER K., HAVEMANN S.: Subdivision surface tessellation on the fly using a versatile mesh data structure. *Computer Graphics Forum* 19, 3 (Aug. 2000), C151–C159. (Proc. Eurographics '2000 Conf.). 39
- [MQ02] McDONNELL K. T., QIN H.: Dynamic sculpting and animation of free-form subdivision solids. *The Visual Computer* 18, 2 (2002), 81–96. 23, 161
- [MQV98] MANDAL C., QIN H., VEMURI B.: Direct manipulation of butterfly subdivision surfaces : A physics-based approach, 1998. 23
- [MQV00] MANDAL C., QIN H., VEMURI B.: A novel fem-based dynamic framework for subdivision surfaces. *Computer-Aided Design* 32 (August 2000), 479–497. (Special issue on solid modeling). 23
- [MQW01] McDONNELL K. T., QIN H., WLODARCZYK R. A.: Virtual clay: A real-time sculpting system with haptic toolkits. In *Proc. ACM Symposium on Interactive 3D Graphics 2001* (March 2001), pp. 179–190. 23
- [MSF99] MÜLLER G., SCHÄFER S., FELLNER D.: Automatic creation of object hierarchies for radiosity clustering. In *Proc. Pacific Graphics '99* (Seoul, Korea, Oct 1999), Kim M.-S., Seidel H.-P., (Eds.), pp. 21–29. 39
- [MSF00] MÜLLER G., SCHÄFER S., FELLNER D. W.: Automatic creation of object hierarchies for radiosity clustering. *Computer Graphics Forum* 19, 4 (Dec 2000), 213–221. 34
- [MTF03] MÜLLER K., TECHMANN T., FELLNER D.: Adaptive ray tracing of subdivision surfaces. *Computer Graphics Forum* 22, 3 (2003), 535–562. Proceedings of Eurographics 2003. 36, 115
- [MTP*04] MCCOOL M., TOIT S. D., POPA T., CHAN B., MOULE K.: Shader algebra. *ACM Trans. Graph.* 23, 3 (2004), 787–795. 158
- [NK03] NOVOTNI M., KLEIN R.: 3d zernike descriptors for content based shape retrieval. In *The 8th ACM Symposium on Solid Modeling and Applications* (June 2003). 9
- [NSACO05] NEALEN A., SORKINE O., ALEXA M., COHEN-OR D.: A sketch-based interface for detail-preserving mesh editing. In *Accepted for publication in Proceedings of SIGGRAPH 2005* (2005), ACM Press. 24
- [Nvi04] NVIDIA CORPORATION: *Nvidia GPU Programming Guide*, May 2004. Version 2.0.1. 130
- [O*] OUSTERHOUT J., ET AL.: Tcl/tk scripting language website. www.tcl.tk. 36
- [OBA*03] OHTAKE Y., BELYAEV A., ALEXA M., TURK G., SEIDEL H.-P.: Multi-level partition of unity implicits. *ACM Transactions on Graphics* 22, 3 (July 2003), 463–470. 19
- [Off03] OFFEN L.: *Interaktiv unterstützte Reparatur nicht mannigfaltiger BReps und Constructive Solid Geometry*. Master's thesis, Institute für ComputerGraphik, Technische Universität Braunschweig, 2003. 41
- [Opea] Opencascade 3d modeling and numerical simulation software development platform. OpenCascade S.A. available from www.opencascade.org. 133
- [Opeb] OpenSG project website. www.opensg.org. 43, 263
- [Opec] OpenSGPLUS project website. www.opensg.org/OpenSGPLUS. 43, 263
- [OS04] OH J.-Y., STUERZLINGER W.: A system for desktop conceptual 3d design. *Virtual Reality, journal, Springer Verlag* 7, 3-4 (Jun 2004), 198–211. 26
- [PA04] PASKO A., ADZHIEV V.: Function-based shape modeling: mathematical framework and specialized language. In *Automated Deduction in Geometry*, Winkler F., (Ed.), Lecture Notes in Artificial Intelligence 2930. Springer-Verlag, Berlin Heidelberg, 2004, pp. 132–160. available from cis.k.hosei.ac.jp/~F-rep. 19, 28
- [Pao] PAOLUZZI: Plasm language homepage. www.plasm.net. 28
- [PASS95] PASKO A. A., ADZHIEV V., SOURIN A., SAVCHENKO V. V.: Function representation in geometric modeling: concepts, implementation and applications. *The Visual Computer* 11, 8 (1995), 429–446. 28, 34

- [PF01] PERRY R. N., FRISKEN S. F.: Kizamu: A system for sculpting digital characters. In *Proceedings of ACM SIGGRAPH 2001* (Aug. 2001), Computer Graphics Proceedings, Annual Conference Series, pp. 47–56. 21
- [PG04] PFISTER H., GROSS M.: Point-based computer graphics. *IEEE Comput. Graph. Appl.* 24, 4 (2004), 22–23. 19
- [PGZF01] POLLEFEYS M., GOOL L. J. V., ZISSERMAN A., FITZGIBBON A. W. (Eds.): *3D Structure from Images - SMILE 2000, Second European Workshop on 3D Structure from Multiple Images of Large-Scale Environments Dublin, Ireland, July 12, 2000, Revised Papers* (2001), vol. 2018 of *Lecture Notes in Computer Science*, Springer. 8
- [PJM94] PRUSINKIEWICZ P., JAMES M., MĚCH R.: Synthetic topiary. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1994), ACM Press, pp. 351–358. 25
- [PKKG03a] PAULY M., KEISER R., KOBBELT L., GROSS M.: Shape modeling with point-sampled geometry. In *Proc. SIGGRAPH 2003* (2003), pp. 641–650. 161
- [PKKG03b] PAULY M., KEISER R., KOBBELT L. P., GROSS M.: Shape modeling with point-sampled geometry. *ACM Trans. Graph.* 22, 3 (2003), 641–650. 22
- [PM01] PARISH Y. I. H., MÜLLER P.: Procedural modeling of cities. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), ACM Press, pp. 301–308. 27
- [png04] Portable network graphics (png) standard. ISO/IEC, 2004. ISO/IEC 15948:2004, available from www.w3.org/Graphics/PNG, also see www.iso.org. 31
- [PPS04] PAOLUZZI A., PASCUCCI V., SCORZELLI G.: Progressive dimension-independent boolean operations. In *Proceedings of the 9th ACM Symposium on Solid Modeling and Applications* (Genova, Italy, June 2004), pp. 203–212. 28
- [PPV95] PAOLUZZI A., PASCUCCI V., VICENTINO M.: Geometric programming: a programming approach to geometric design. *ACM Trans. Graph.* 14, 3 (1995), 266–306. 28, 34
- [Pra04] PRATT M.: Extension of iso 10303, the step standard, for the exchange of procedural shape models. In *Proc. International Conference on Shape Modeling and Applications (SMI04)* (Genova, Italy, June 2004), pp. 317–326. 12, 13, 29
- [PS85] PREPARATA F. P., SHAMOS M. L.: *Computational Geometry: An Introduction*. Springer, New York, 1985. 25
- [PS96] PULLI K., SEGAL M.: *Fast Rendering of Subdivision Surfaces*. Tech. rep., University of Washington, 1996. 39
- [PT00] PIZER S., THALL A.: M-reps: A new object object representation for graphics. *ACM Transactions on Graphics* (submitted 1999), 2000. 22
- [QT95] QIN H., TERZOPOULOS D.: Dynamic nurbs swung surfaces for physics-based shape design. *Computer-aided Design* 27, 2 (1995), 111–127. 23
- [Qua] Quake (ego shooter game and game engine). Id Software Inc. www.idsoftware.com. 262
- [R*] RECHLIN E., ET AL.: Hp calculator open interest group website. www.hpcalc.org. 216
- [RA99] RAMAMOORTHY R., ARVO J.: Creating generative models from range images. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1999), ACM Press/Addison-Wesley Publishing Co., pp. 195–204. 30
- [RDR03] REPPY J., DANI V., RIEHL J.: *Project 1 – Ray Tracing, and Handout 3 – The GML Specification*. Course cmsc 23700 ‘introduction to computer graphics’, fall 2003, University of Chicago, Oct 2003. available from www.classes.cs.uchicago.edu. 51

- [Ror86] RORICZER M.: Büchlein von der fialen gerechtigkeit. Staatliche Bibliothek Regensburg, Germany, 1486. [229](#)
- [RS97] RAPPOPORT A., SPITZ S.: Interactive boolean operations for conceptual design of 3-d solids. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1997), ACM Press/Addison-Wesley Publishing Co., pp. 269–278. [25](#)
- [RVB02] REINERS D., VOSS G., BEHR J.: OpenSG - basic concepts. In *Proceedings of OpenSG Symposium 2002* (2002). [43](#), [263](#)
- [Sch00] SCHÄFER S.: *Efficient object-based hierarchical radiosity methods*. PhD thesis, Technische Universität Braunschweig, 2000. available as electronic dissertation from www.biblio.tu-bs.de. [39](#)
- [SCOT03] SORKINE O., COHEN-OR D., TOLEDO S.: High-pass quantization for mesh encoding. In *Proc. Symp. on Geometry Processing (SGP'03)* (Aachen, Germany, 2003), Eurographics/ACM SIGGRAPH, pp. 42–51. [20](#)
- [Sei91] SEIDEL R.: A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry: Theory and Applications 1*, 1 (1991), 51–64. [157](#)
- [SG72] STINY G., GIPS J.: Shape grammars and the generative specification of painting and sculpture. In *Information Processing 71*, Freiman C. V., (Ed.). North Holland, Amsterdam, 1972, pp. 1460–1465. [27](#)
- [SG03] SURAZHISKY V., GOTSMAN C.: Explicit surface remeshing. In *SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing* (Aachen, Germany, 2003), Eurographics/ACM SIGGRAPH, pp. 20–30. [20](#)
- [SH05] SU W. Y., HART J. C.: A programmable particle system framework for shape modeling. In *Proc. Shape Modeling International* (MIT, Cambridge (MA), USA, June 2005). [22](#)
- [Sha02] SHAPIRO V.: Solid modeling. In *Handbook of Computer Aided Geometric Design*. Elsevier, 2002, ch. 20, pp. 473–518. [12](#), [13](#)
- [SK92] SNYDER J. M., KAJIYA J. T.: Generative modeling: A symbolic system for geometric modeling. In *SIGGRAPH 92 Conference Proceedings* (july 1992), ACM SIGGRAPH, pp. 369–378. [29](#), [34](#)
- [SL95] STEPANOV A., LEE M.: *The Standard Template Library*. Hewlett-Packard Laboratories, 1995. [40](#), [137](#), [139](#), [155](#)
- [SM78] STINY G., MITCHELL W. J.: The palladian grammar. *Environment and Planning B*, 5 (1978), 5–18. [27](#)
- [Sny92] SNYDER J. M.: *Generative Modeling for Computer Graphics and CAD*. Academic Press, San Diego, CA, 1992. [29](#), [212](#)
- [Sou] Open source software development website. SourceForge (VA Systems). at sourceforge.net. [17](#), [133](#)
- [SPS01] SCHKOLNE S., PRUETT M., SCHRÖDER P.: Surface drawing: creating organic 3d shapes with the hand and tangible tools. In *CHI '01: Proceedings of the SIGCHI conference on Human factors in computing systems* (New York, NY, USA, 2001), ACM Press, pp. 261–268. [24](#)
- [SPS04] SCHMITT B., PASKO A., SCHLICK C.: Constructive sculpting of heterogeneous volumetric objects using trivariate b-splines. *The Visual Computer* 20, 2-3 (2004), 130–148. [19](#)
- [SQL91] Database language sql. ISO ANSI, 1991. ISO/IEC 9075:1992, available from www.iso.org. [267](#)
- [sql96] Sql2 standard. ISO/IEC, 1996. JTC1/SC21 N10489, ISO/IEC 9075, Committee Draft (CD), Database Language SQL, Part 2: SQL/Foundation, see www.iso.org. [267](#)
- [SR04] SPITZ S., RAPPOPORT A.: Integrated feature-based and geometric cad data exchange. In *Proceedings of the 9th ACM Symposium on Solid Modeling and Applications* (Genova, Italy, June 2004), pp. 183–190. [11](#)
- [SSS*02] SUTANTHAVIBUL S., SATO T., SMITH B., ET AL.: Xfig user manual version 3.2.4, Dec 2002. open source interactive drawing tool, available from www.xfig.org. [260](#)
- [Sta98] STAM J.: Exact evaluation of catmull-clark subdivision surfaces at arbitrary parameter values. In *Proceedings of SIGGRAPH 98* (1998), pp. 395–404. [101](#), [115](#)

- [Sta99] STAM J.: Evaluation of loop subdivision surfaces. ACM SIGGRAPH 2000 Course #20 Notes, 1999. 101, 115
- [ste94] Iso 10303, standard for the exchange of product model data (step), officially ‘industrial automation systems and integration – product data representation and exchange’. ISO/IEC, 1994. available from www.iso.org. 13
- [SVG03] Scalable vector graphics (svg). World Wide Web Consortium, 2003. Language Specification 1.1 W3C recommendation available from www.w3c.org. 15
- [SVY99] SCHIRRA S., VELTKAMP R., YVINEC M.: *CGAL Reference Manual, Release 2.0*. University of Utrecht, Netherlands, 1999. available from www.cgal.org. 133
- [SWG*03] SANDER P. V., WOOD Z. J., GORTLER S. J., SNYDER J., HOPPE H.: Multi-chart geometry images. In *Proc. Symp. on Geometry Processing (SGP’03)* (Aachen, Germany, 2003), Eurographics/ACM SIGGRAPH, pp. 146–155. 20
- [SZSS98] SEDERBERG T. W., ZHENG J., SEWELL D., SABIN M.: Non-uniform recursive subdivision surfaces. In *Proc. SIGGRAPH 1998* (1998), ACM Press, pp. 387–394. 161
- [TC1] Organizational webpage. TC184/SC4. www.tc184-sc4.org. 13
- [TDT98] T. DEROSE M. K., TRUONG T.: Subdivision surfaces in character animation. In *Proceedings of SIGGRAPH 98* (1998), pp. 85–94. 35, 36, 41, 91, 112
- [TG98] TOUMA C., GOTSMAN C.: Triangle mesh compression. In *Proc. Graphics Interface (GI ’98)* (1998), pp. 28–34. 82
- [the99] The matrix (movie). Andy Wachowski and Larry Wachowski, 1999. 17
- [TPF00] THALL A., PIZER S., FLETCHER T.: *Deformable Solid Modeling using Sampled Medial Surfaces: A Multiscale Approach*. Tech. rep., Chapel Hill, NC, USA, 2000. 22
- [Tur36] TURING A. M.: On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.* 2, 42 (1936), 230–265. 30, 57
- [U3D04] U3d universal 3d file format for downstream 3d cad repurposing and visualization purposes. 3D Industry Forum, 2004. ECMA-363 standard available from www.3dif.com. 16
- [UM04] UNGEWITTER G., MOHRMANN K.: *Lehrbuch der gotischen Konstruktionen*, 4 ed., vol. 1–4. Chr. Herm. Tauchnitz, Leipzig, 1904. 229
- [v*] VAN ROSSUM G., ET AL.: Python scripting language website. www.python.org. 215
- [Vel99] VELTKAMP R.: Generic geometric programming in the computational geometry algorithms library. *Computer Graphics Forum* 18, 2 (1999), 131–137. 133
- [Vie] 3d model collection homepage. Viewpoint Corporation. available from www.viewpoint.com, in 2002 migrated to www.digimation.com. 109
- [VKK*03] VARADHAN G., KRISHNAN S., KIM Y. J., DIGGAVI S., MANOCHA D.: Efficient max-norm distance computation and reliable voxelization. In *Proc. Symp. on Geometry Processing (SGP’03)* (Aachen, Germany, 2003), Eurographics/ACM SIGGRAPH, pp. 116–126. 20
- [Vog05] VOGEL H.: Autodesk übernimmt compass. *c’t magazin für computertechnik*, 6 (March 2005), 68. 14
- [VPBY02] VELHO L., PERLIN K., BIERMANN H., YING L.: Algorithmic shape modeling with subdivision surfaces. *Computers and Graphics* 26 (December 2002), 865. 25
- [VRM97] VrmI, the virtual reality modeling language. Web3D Consortium, 1997. Language Specification Version 2.0, ISO/IEC IS 14772-1, available from www.web3D.org. 15, 213
- [VTMH04] VARLEY P., TAKAHASHI Y., MITANI J., H.SUZUKI: A two-stage approach for interpreting line drawings of curved objects. In *First Eurographics Workshop on Sketch-Based Interfaces and Modeling (SBM’04)* (Grenoble, France, sep 2004), Hughes J., Jorge J., (Eds.), Eurographics, pp. 117–126. 24

- [WH94] WITKIN A. P., HECKBERT P. S.: Using particles to sample and control implicit surfaces. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1994), ACM Press, pp. 269–277. 22
- [wika] Reverse polish notation (rpn). Wikipedia.org. en.wikipedia.org/wiki/Rpn. 216
- [wikb] Solid modeling. Wikipedia.org. www.wikipedia.org/wiki/Solid_modeling. 11
- [WLM00] WONG J., LAU R., MA L.: Virtual 3d sculpting. *Journal of Visualization and Computer Animation* 11, 3 (July 2000), 155–166. 24
- [WMKE04] WEILER M., MALLÓN P. N., KRAUS M., ERTL T.: Texture-Encoded Tetrahedral Strips. In *Proceedings Symposium on Volume Visualization 2004* (2004), IEEE, pp. 71–78. 19
- [WND97] WOO M., NEIDER J., DAVIS T.: *OpenGL Programming Guide – The Official Guide to Learning OpenGL, Version 1.1*, 2nd ed. Addison Wesley Developers Press, 1997. 17, 84
- [WS01] WESCHE G., SEIDEL H.-P.: Freedrawer: a free-form sketching system on the responsive workbench. In *VRST '01: Proceedings of the ACM symposium on Virtual reality software and technology* (New York, NY, USA, 2001), ACM Press, pp. 167–174. 24
- [WTL*04] WANG X., TONG X., LIN S., HU S., GUO B., SHUM H.-Y.: Generalized displacement maps. In *Rendering Techniques 2004: 15th Eurographics Workshop on Rendering* (June 2004), pp. 227–234. 25
- [WW94] WELCH W., WITKIN A.: Free-form shape design using triangulated surfaces. *Computer Graphics* 28, Proc. SIGGRAPH '94 (1994), 247–256. 23, 161
- [WW02] WARREN J., WEIMER H.: *Subdivision Methods for geometric design*. Morgan Kaufmann Publishers, 2002. 91, 96
- [WWSR03] WONKA P., WIMMER M., SILLION F. X., RIBARSKY W.: Instant architecture. *ACM Transactions on Graphics* 22, 3 (July 2003), 669–677. 27
- [X3d03a] Extensible 3d (x3d) graphics working group. Web3D Consortium, 2003. available from www.web3D.org. 213
- [X3D03b] X3d framework and sai (scene access interface). ISO/IEC, 2003. FDIS (final draft international standard) 19775:200x, available from www.web3d.org. 213
- [X3D04] X3d, extensible 3d markup language. Web3D Consortium, 2004. Language Specification ISO/IEC FDIS 19775-1:200x available from www.web3D.org. 15
- [XML] Xml specifications. W3C World Wide Web Consortium. www.w3.org/XML. 267
- [XSL] The extensible stylesheet language family (xsl). W3C World Wide Web Consortium. www.w3.org/Style/XSL. 267
- [ZHH96] ZELEZNIK R. C., HERNDON K. P., HUGHES J. F.: Sketch: an interface for sketching 3d scenes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), ACM Press, pp. 163–170. 24
- [ZPKG02] ZWICKER M., PAULY M., KNOLL O., GROSS M.: Pointshop 3d: an interactive system for point-based surface editing. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2002), ACM Press, pp. 322–329. 22
- [ZS99] ZORIN D., SCHROEDER P.: Subdivision for modeling and animation. ACM SIGGRAPH 1999 Course Notes, 1999. 91, 92, 96
- [ZSS97] ZORIN D., SCHRÖDER P., SWELDENS W.: Interactive multiresolution mesh editing. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1997), ACM Press/Addison-Wesley Publishing Co., pp. 259–268. 23, 162

List of Tables

- 3.1 Vertex classification according to the number of incident sharp edges 105
- 3.2 Face classification according to the vertex classification and edge types. 105
- 3.3 Grid sizes per level 122
- 3.4 Normal variation over the patch vs. variation over the first quad only 130

List of Figures

1.1	Lego pieces	3
1.2	Structural similarity	4
1.3	Structural similarity	4
1.4	Structural similarity	5
1.5	Structural similarity	5
1.6	Examples of the Maya Embedded Language (MEL) from Alias/Wavefront	10
1.7	Typical procedural modeling tools and high-end CAD systems	11
1.8	3D Presentation software and commercial game engines	14
1.9	Open source 3D engines	17
1.10	Profile product in GENMOD	30
1.11	The classical workflow in any public or scientific library	33
1.12	Results from the diploma thesis on <i>Generative Modeling</i>	34
1.13	The Python-based subdivision surface modeling toolkit	35
1.14	Gallery of models made with the Python modeling toolkit	36
1.15	The power of high-level tools	37
1.16	Lego-style modeling with subdivision surfaces	37
1.17	Python code examples	38
1.18	Parametric triangle meshes.	39
1.19	Modeling combined B-reps with Slick	39
1.20	Gallery of Slick models	40
1.21	CSG on arbitrary meshes by Lars Offen	41
1.22	Lego.	42
1.23	The Charismatic workflow for the Wolfenbüttel reconstruction	43
1.24	Charismatic: comparison between real and virtual Wolfenbüttel	44
1.25	Charismatic: Wolfenbüttel houses	44
1.26	Viewer applications in the DAVE	45
1.27	Physical setup of the DAVE	47
1.28	Initial problems with the DAVE.	48
1.29	Three generations of web plugins	49
1.30	Turbine made with <i>Slick</i> .	50
1.31	GML example for modeling with a stack-based language	51
1.32	Gothic architecture example	52
1.33	Available IDE's for GML.	53
1.34	Parameterized model of a chair	55
2.1	Example of an identification space	62
2.2	Cell complex on a sphere	63
2.3	Different examples of cell complexes	64
2.4	Cylinder vs. Moebius band	64
2.5	Torus construction	65
2.6	Klein bottle	66
2.7	Connected sum of two tori	66
2.8	Planar diagrams of sphere and crosscap	67
2.9	Genus and Euler characteristic	67
2.10	Directed Complex and Ring Examples	68
2.11	Paths on a double torus	68

2.12	Platonic Solids	69
2.13	Planar models of Euler operators	70
2.14	Conditions for legal application of Euler Operators	71
2.15	Euler Operator Example: Quadrangular Torus	73
2.16	killFmakeRH operation from below	74
2.17	Algebraic properties of Euler Operators.	75
2.18	Euler coordinates of a mesh.	76
2.19	Problems with Klein Bottle and killFmakeRH	78
2.20	Removing a double loop	79
2.21	Mesh Removal using Euler operators	79
2.22	Self-intersections are sometimes hard to detect	81
2.23	Indexed Face Set file formats	83
2.24	Indexed Face Set in OpenGL	83
2.25	OpenGL display primitives	84
2.26	Swapping OpenGL triangle strips	84
2.27	Inflating the Möbius band	85
2.28	Possible Pitfalls with Polygonal Meshes.	86
2.29	Inflating neighbouring face and ring.	87
2.30	Patch complex examples	89
3.1	Knot intervals mapped to B-spline control polygon	92
3.2	B-Spline functions of degree 0 and 1	93
3.3	B-spline functions of degree 2 and 3	93
3.4	De Boor algorithm, general version	94
3.5	De Boor algorithm, cubic version	94
3.6	B-spline subdivision	95
3.7	B-spline subdivision	96
3.8	Subdivision of B-spline surface	98
3.9	Regular subdivision stencils	98
3.10	Example of recursive subdivision	99
3.11	Stencil for limit position and tangents	100
3.12	Limit points on a cube	101
3.13	Crease curve design	102
3.14	Stencil for crease limit position and tangents	103
3.15	Simple example of an object with a crease	104
3.16	Vertex and Face type examples.	105
3.17	Influence of a single base vertex	106
3.18	Face interpolation	106
3.19	Subdivision surface of a zero area face	107
3.20	Shift of the face centroid	107
3.21	Many vertices attract the surface	108
3.22	Foldover example	108
3.23	Linear limit	109
3.24	Non-convex faces	109
3.25	Ripples with high valences	110
3.26	The Viewpoint models and their subdivision surfaces	111
3.27	Complete vs. incomplete refinement	114
3.28	Basis functions of a regular patch.	116
3.29	Maple code for direct weights	117
3.30	Vertex and Face Rings	120
3.31	Patch Subdivision Levels.	120
3.32	Limit projection pattern	121
3.33	Subdivision and limit projection patterns.	122
3.34	Block subdivision rule	123
3.35	Tile Limit Projection Rule	124
3.36	Part of the patch tessellation function.	125
3.37	Adaptive refinement towards the patch borders	126

3.38	OpenGL calls to render a patch	127
3.39	Adaptive display of subdivision surfaces	128
3.40	The final rule	129
3.41	Operation counts for regular stencils.	129
3.42	Rule application counts	129
3.43	Number of rule applications	129
3.44	Extreme bending of a subdivision patch	130
3.45	Shading artifacts	131
3.46	Subdivision surfaces approximating a sinoidal wave.	131
3.47	Hierarchy of fixed-size tessellations	132
3.48	Adaptive display of real subdivision surfaces	132
4.1	Triangle grid	134
4.2	Shared vertex triangle mesh, static version	135
4.3	Gathering connectivity information	136
4.4	Shared vertex implementation using traits	138
4.5	Relocation repair function	138
4.6	Iterating over a triangle mesh	141
4.7	Directed Edge triangle mesh	142
4.8	The wedge problem	143
4.9	Two forms of pair contraction	144
4.10	Edge collapse & vertex split, and geomorph	147
4.11	Simplification of scanned data	148
4.12	Adaptive level of detail with progressive meshes	148
4.13	Simplification implies smoothing	148
4.14	CAD detail reduction	149
4.15	Detail preservation	150
4.16	Simplification breaks symmetry	150
4.17	B-rep connectivity example	151
4.18	Red artifact edges to split up rings	152
4.19	BRep mesh data structure	153
4.20	Some B-rep functions.	153
4.21	Design decisions of class BRepMesh	154
4.22	Skipvector and Skipchunk interfaces	156
4.23	Mehlhorn triangulation	157
4.24	Triangulation API	158
4.25	Rendering a triangulated B-rep	159
4.26	Combined B-rep Example: Arcade	160
4.27	Combined B-rep Example: Temple	161
4.28	Combined B-rep Examples: Non-planar Faces and Tunnel	162
4.29	Combined B-rep example: Ornamental detail	163
4.30	Combined B-rep example: Levels of window detail	163
4.31	Combined B-rep example: Window decorations	163
4.32	Combined B-rep classification example	164
4.33	Combined B-rep trait	165
4.34	Multi-triangulation of sharp faces	166
4.35	One patch per halfedge	167
4.36	Illustration of the data in a patch structure	168
4.37	Catmull/Clark tessellation class	169
4.38	Catmull/Clark patch control mesh from vertex and face rings	169
4.39	Catmull/Clark tessellation	170
4.40	Different cases of vertex rings	171
4.41	Apex translation of the view cone	174
4.42	Back-patch culling using the normal cone	175
4.43	The projected size heuristic	176
4.44	Excess of the view cone at grazing angles	176
4.45	The curvature and crease heuristics	176

4.46	Tessellation of smooth faces	178
4.47	Mesh manipulation through Euler operators	180
4.48	The Progressive B-Rep class	181
4.49	Data in log records of extended Euler operators	182
4.50	The dependency relation of Euler macros	184
4.51	Inconsistent intermediate configurations.	185
4.52	Euler macro hierarchy example	186
4.53	Macro culling examples	188
4.54	poly2doubleface, simple version	190
4.55	Extrusion illustrated	190
4.56	extrude, simple version	190
4.57	bridgeFaces, simple version	190
4.58	Bridging two faces	190
4.59	Computing the vertices of the offset polygon	192
4.60	Polygon scaling vs. edge offsets.	192
4.61	Sharpness modes for extrusion	193
4.62	Column basis created by multiple extrusion	194
4.63	The principle of path extrusion	195
4.64	Parameters of path extrusion	195
4.65	Typical façade profiles created with path extrusion	196
4.66	Reflection property	196
4.67	The straight skeleton	197
4.68	Examples of the straight skeleton	198
4.69	Straight skeleton of points on a circle	199
4.70	The medial axis of a polygon	199
4.71	Intersection-free offset polygon at distance $d > 0$	200
4.72	The roof of a castle	201
4.73	Comparison between polygon offset and straight skeleton	202
4.74	Subdivision surface from irregular straight skeleton	202
4.75	The offset operation is not always invertible	203
4.76	Flipping bisector angles due to bowtie offsets	204
4.77	Reversible multiple offsets	204
4.78	Two ways to build a tower with four corner towers	205
4.79	Ornamental profile using reversible offset.	206
4.80	Tubular shapes through intersection-free extrusion with a half-circle profile	207
4.81	Problem with intelligent modeling tools.	208
5.1	Expressing apparent structural similarity	212
5.2	Requirements for an underlying shape representation	214
5.3	Layered GML software architecture	214
5.4	Tokenizer patterns for literal tokens	216
5.5	The twelve language rules of the GML	217
5.6	Essential GML operators	218
5.7	Step by step execution of GML code	219
5.8	Basic modeling techniques	221
5.9	A simple way to create a solid	221
5.10	Designing a loop step by step	222
5.11	A stack item is put aside using dup	222
5.12	Segment intersection followed by offset polygon	223
5.13	Euler operators as GML operators	224
5.14	Building a cube with Euler operators	225
5.15	Creating a re-usable arch or door	226
5.16	A door in a wall as a function	226
5.17	Dom	229
5.18	The Braunschweig city hall	230
5.19	Highly coded construction plans	230
5.20	Style and structure	231

5.21	Stone masons at work	231
5.22	Evolution of Gothic window tracery	232
5.23	Pointed arch and round arch	233
5.24	International success of Gothic architecture	233
5.25	Geometry of the prototype window	234
5.26	The fields within a Gothic window	234
5.27	Construction of a rosette	235
5.28	Window tracery with flat and round bars	236
5.29	Basic and sophisticated rosettes	236
5.30	Parameters of the standard Gothic window	237
5.31	Using the Gothic window modeling tool.	241
5.32	The style library for the simple style 1	241
5.33	Library of the doubly recursive style 8	241
5.34	Input parameters of the style functions	241
5.35	Comparison of styles 1 and 8	241
5.36	Main window function gw-gothic-window	242
5.37	Function gw-pointed-arch	242
5.38	Function gw-polygon-2arcs-height	242
5.39	Function gw-compute-arcs-rosette	242
5.40	Five basic Gothic window styles	243
5.41	Parameter variations in a manifold of Gothic windows	244
5.42	The principle of information unfolding applied	245
5.43	Dom	247
5.44	Fotos from the Cologne cathedral	248
5.45	Dom	249
5.46	Entries of a pillar dictionary	250
5.47	create-pillars	251
5.48	create-walls	251
5.49	build-wall	251
5.50	Part of the function set-type-pillars	252
5.51	Part of the function set-type-walls	252
5.52	make-wall is called by set-type-walls in Fig. 5.51 above	252
5.53	make-pillar	252
5.54	getcol	252
5.55	Pillar-Nave as a pillar example; set as pillar type of rows 4,5 in set-type-pillars in Fig. 5.50 above	252
5.56	The eight different pillar types	253
5.57	The different wall types	253
5.58	Dom transparent	254
5.59	GML dictionary as object-oriented class	256
5.60	Versatility of the Postscript syntax	257
5.61	Versatility of the Postscript syntax	258
5.62	Comparison of the different complexity classes	259
5.63	The GML for creating interactive images and	260
5.64	GML implementation	261
5.65	The interactive CAVE planner	261

Index

Symbols

k -chains	68
k -cycles	69
n -cell	63
V^3D^2	33
“Verteilte Vermittlung und Verarbeitung Digitaler Dokumente”	33

A

AABBBox	187
abstract 2-complex	80
abstract 3D object	262
adaptive rendering	88
adaptively sampled distance field	21
algebra of chains	68
AoS vs. SoA issue	126
apex translation	174
application-specific operators	264
artifact edges	152
artist	9
associative design	11
attachment	2

B

B-rep	18
Bézier curve	93
base edge	100
base face	80, 100
base mesh	100
base vertex	100
basemesh	146
behaviour graph	14
Betti numbers	68
blend tree	19
boundary operator	68
boundary representation	151
bowtie	86
brushes	25, 262

C

Cave	45
cB-reps	164
CCW	64
CCW orientation	136
cell	25
cell complex	63
Charismatic	43
child macro	184
child of a base vertex	100

classification of surfaces theorem	66
closedness of Euler operators	76
code generation problem	12
combined B-rep	160
compact set	61
compatible signatures	222
completeness of Euler operators	78
complex edge	86
complex vertex	86
compositing	9
computer-aided design	11
connected sum of surfaces	65
connectivity of a mesh	80
consistent orientation	64
constraints	11, 12
Constructive Solid Geometry	41
container data structure	40
container data structures	137
containment	2
control mesh	100
control mesh surface	106
convex differences aggregates	25
corner	144
corner vertex rule	104
crease edge rule	103
crease in a surface	143
crease normal rule	103
crease vertex rule	103
crosscap	67
crossing piers	250
CSG	60
Culling	88
current macro	183
CW	64

D

dangling vertex	71
dart vertex rule	104
de Boor algorithm	93
definition of a feature	12
degenerate triangle	141
degree 2 face	86
delta encoding	228
design by features	12
design space	21
designer	9
detail textures	9
Digital Libraries	33

- dimensioning 11
 directed edge 142
 directed edge data structure 142
 discrete topology 60
 dissolve 229
 DL 33
 double loop 78, 79
 double loops 79
 dual complex 70
 dynamic signature 223
- E**
 edge collapse 71, 146
 edge cycle 67
 edge cycle of a face 70
 edge rule 99
 edges 63
 embedding 80
 equidistant knot vector 92
 Euclidean distance 61
 Euler characteristic 67
 Euler macros 41, 183
 Euler operators 41, 70
 exit 225
 expressions 12
 extended Euler operators 179
 External Authoring Interface 15
 extrinsic 259
 extrusion 72
- F**
 face degree 63
 face depth 127
 face join 71
 face rule 99
 face, degree 1 86
 faces 63
 features 12
 fields 15
 finite element method 23
 floor height 11
 fold-overs 108
 function, continuous 61
 function, invertible 61
- G**
 generalized 3D documents 34
 generalized document 33
 generative linguistics 27
 generative model 29
 Generative Shape 26
 generic programming 137
 genus 67
 geometric constraints 12
 geometric continuity 35, 89
 geometrical consistency 81
 geometrically degenerate 141
 gizmos 260
- gluing 62
 GML approach 7
 GML efficiency conjecture 56
 graphical user interface 36
- H**
 halfedge 140
 halfedge index 142
 Hausdorff space 60
 hierarchical meshes 88
 hollow faces 75
 homeomorphism 61
 homology 68
 human perception 81
- I**
 ID size 135
 identification space 62
 IFS, indexed face sets 82
 incomplete refinement 114
 index array 85
 indexed face set 82
 inflate a model 87
 intelligent 3D components 11
 internal Euler operators 183
 intrinsic 259
 intrinsic property 61
 isomorphism 62
 iterator 141
- K**
 Kinsey, L. Christine 59, 61, 66, 68
 Kizamu 21
 Klein bottle 66
 knot insertion 99
 knot insertion for B-spline curves 94
- L**
 Lego 2
 level set 21
 local control mesh 121
 locally planar graph 70
 Looking Glass 265
 loop 86
- M**
 Mäntylä, Martti 59, 60, 78, 86
 Möbius band 64
 Möbius band 64
 makeEF 71
 makeEkillR 71
 makeEV 71
 makeFkillRH 71
 makeVEFS 179
 makeVFS 71, 179
 manifold 65
 manifold complex 65
 manifold edge property 67

- manifold vertex property 67
 - manifolds 35
 - Maya Embedded Language 10
 - memory fragmentation 155
 - mesh connectivity 80
 - mesh compression 31
 - mesh connectivity rule 99
 - mesh consistency theorem 67
 - mesh geometry 80
 - mesh simplification 31
 - mesh, deleting a mesh 78
 - mesh, valid 67
 - meta-rule 3
 - metaballs 19
 - metric 61
 - metric space 61
 - metric topology 61
 - Minimal Rendering Toolkit 39
 - modeler 9
 - moving least squares 22
 - MRT B-reps 39
 - multi-resolution mesh 146
 - multi-view photograph 8
 - multiple edges 86
 - multiple extrusions 194
 - multiple vertex 86
 - multiplicity of a knot 94
- N**
- nature of shape 59
 - node types 19
 - non-coplanar ring 86
 - non-orientable edge 86
 - non-orientable surface 64
 - non-photorealistic rendering 24
 - non-planar face 86
 - non-simple face boundary 86
 - normal cone technique 175
 - null edge 86
- O**
- open ball 61
 - Open sets 60
 - operation count 123
 - orientability of a 2-complex 67
 - orientation, consistent 64
 - outside ring 86
- P**
- parametric design 11
 - parametric patch 88
 - parent macro 184
 - partition of unity implicits 19
 - patch 106
 - patch complex 88
 - path expressions 141
 - picking problem 13
 - planar diagram 63
 - planar models of Euler operators 70
 - plane models 69
 - Platonic solids 69, 70
 - PM 146
 - pointed sphere 61, 63
 - Pointshop3D 22
 - polygonal face 105
 - polyhedron 65
 - polymorphic 223
 - PostScript 50
 - principal planes 159
 - Principle of Information Unfolding 26
 - principle of information unfolding 246
 - projective plane 66
 - psychology of a computer 81
- Q**
- quasi regular 4×4 grid 121
 - quotient topology 62
- R**
- radial basis functions 19
 - redo-direction 183
 - refinement cache 115
 - registration of a 3D scan 8
 - regularized set operations 60
 - relative neighbourhood, subspace topology 62
 - relocation 139
 - relocation problem 139
 - respecting the macro parent-child relation 187
 - reversed orientation 86
 - reversed ring 86
 - ring 80
 - ring in a cell complex 68
 - ring of a direct neighbour face 87
- S**
- S-mesh 36
 - screw 2
 - segmentation problem 8
 - selective updates 132
 - self-intersection 86
 - self-intersections are hard to detect 81
 - semantic constraints 12
 - semantic magnifying glass 187
 - separation axioms 60
 - SFX department 9
 - shader networks 9
 - ShaOLin 36, 115
 - shape matching problem 9
 - shape vocabulary 228
 - shared vertex data structure 135
 - sharp edges 160
 - sharp face 105
 - sharpness flag 160
 - sheet, multiple 86
 - shell 67
 - shells 69

side effect	12
side effects	227
signature of a mesh	75
signed distance function	19
simplex	63
simplicial complex	63
size unit for 32 bit architectures	135
skipvector	155
smooth edges	160
smoothing a B-spline curve	95
smoothing group	143
snap modes	209
solid angle	175
sphere with a zipper	63
standard template library	137
stereographic projection	61
stored procedures	266
stride	159
strip geometry	84
subdivision of a B-spline curve	94
subdivision surfaces	35
support of a cell complex	63
surface	65
surface patch	88
surface shift	107
surfels	25
sweep line	157
sweep line algorithms	157
system shapes	42

T

T-vertex	86
tessellation	88
tessellation on demand	118, 160
Thies, Harmen	229
Thorhauer	229
topological consistency	81
topological dual	70
topological equivalence	61
topological invariant	61
topological space	60
topologically degenerate	141
Topology	60
torus via identification spaces	65
touching vertex	86
triangle budget	175
triangle soups	82
triangle strip list	85
TSL, triangle strip list	85

U

UEA scene graph	43
undo capability	179
undo-direction	183
undo/redo budget	187
union of balls	61

V

valid mesh	67
variant design	12
variation of rules	3
variational class of solids	12
vecadd operation	123
vecmul operation	123
vertex arrays	84
vertex cache	158
vertex rule	99
vertex split	146
vertex valence	63
vertices	63
view cone	174
virtual	7
visitor	14
visual importance	177
voxel	61
VRML node/field	19

W

wedge	119, 144
window tracery	228