INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2022IPPAT027

Thèse de doctorat

Interactive Authoring of 3D Shapes Represented as Programs

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom Paris

École doctorale n°626
École Doctorale de l'Institut Polytechnique de Paris (ED IP Paris)
Spécialité de doctorat : Signal, Images, Automatique et Robotique

Thèse présentée et soutenue à Palaiseau, le 11/07/2022, par

ÉLIE MICHEL

Composition du Jury :

Marie-Paule Cani Professor LIX, CNRS/École Polytechnique	Présidente
Eric Galin Professor LIRIS, CNRS/Université Claude Bernard Lyon 1	Rapporteur
Sylvain Lefebvre Research director Inria Nancy	Rapporteur
Adrien Bousseau Researcher Inria Sophia-Antipolis	Examinateur
Damien Rohmer Professor LIX, CNRS/École Polytechnique	Examinateur
Olga Sorkine-Hornung Professor ETH Zurich	Examinatrice
Tamy Boubekeur Professor Adobe Research	Directeur de thèse

Remerciements

Ma gratitude va bien sûr en premier lieu à mon directeur de thèse, Tamy Boubekeur, qui a su me faire confiance dès notre premier échange, et m'a guidé avec une grande efficacité dans l'expérience de la thèse. Je suis ensuite très reconnaissant envers les différents membres de mon jury d'avoir accepter d'y prendre part, à commencer par les deux rapporteurs, Eric Galin et Sylvain Lefevbre, Adrien Bousseau, Marie-Paule Cani – à qui je dois en particulier ma première entrée en contact avec notre communauté de recherche, Damien Rohmer et Olga Sorkine-Hornung. Je suis sincèrement honoré par la composition de cette assemblée.

Mes remerciements vont ensuite au *Computer Graphics Group* de Télécom Paris, à tous ceux avec qui j'ai pu partager un bureau, ou une de ces pauses si précieuses à la réflexion scientifique, et avec qui je continuerai de partager le souvenir d'une grande convivialité : Chloé Paliard et Alban Gauthier, qui ont accompagné une grande partie de ma thèse, Gilles Laurent, pour ses nombreux éclairages techniques alors que je débutais, Jean-Marc Thiery et Thibaud Lambert, avec qui j'ai noirci quelques tableaux blancs, Jérémie Schertzer, Tong Zhao, Yassine Mankai, Corentin Mercier, Thibault Lescoat, Sylvain Rousseau, Adrien Kaiser, Hélène Legrand, Malik Boughida, Théo Thonat, Victor Lucquin et les permanents Kiwon Um et Amal Dev Parakkat, ainsi que le reste de l'équipe IMAGES.

Je tiens à remercier mes pairs, la communauté de recherche en informatique graphique dans son ensemble, et en particulier sa part, très française, que la récente pandémie a bien voulu me laisser rencontrer, ou celle, très anonyme, qui a assuré des revues toujours justes de mes propositions de publication. La conscience de partager ses questionnements au delà des murs de son bureau est une force de motivation à ne pas sous-estimer.

Je ne pourrais pas oublier de chaleureusement remercier la formation *Arts et Technologies de l'Image*, à l'Université Paris 8, pour toutes les belles rencontres que j'y ai faites au cours des deux années qui ont précédé ma thèse et pour le recul que son enseignement m'a permis d'avoir dans l'appréhension de mon sujet de

thèse. Pour le plongement dans les applications artistiques de mon domaine de recherche, je remercie également tous les membres et sympathisants du collectif Cookie, et toutes celles et ceux qui on voulu croire en ma propre capacité à créer.

Je conclue par saluer celles et ceux que j'ai des raisons de remercier bien au delà du cadre de ce travail. Mes parents, mes frères, et plus généralement ma famille, et mes amis, que je ne vais pas énumérer mais dont la présence, l'écoute, les rires et les riches échanges ont été d'un indispensable soutien. Sachez toutes et tous que vous comptez pour beaucoup.

Abstract

Although hardware and techniques have considerably improved over the years at handling heavy content, digital 3D creation remains fairly complex, partly because the bottleneck also lies in the cognitive load imposed over the designers. A recent shift to higher-order representation of shapes, encoding them as computer programs that generate their geometry, enables creation pipelines that better manage the cognitive load, but this also comes with its own sources of friction. We study in this thesis new challenges and opportunities introduced by program-based representations of 3D shapes in the context of digital content authoring.

We investigate ways for the interaction with the shapes to remain as much as possible in 3D space, rather than operating on abstract symbols in program space. This includes both assisting the creation of the program, by allowing manipulation in 3D space while still ensuring a good generalization upon changes of the free variables of the program, and helping one to tune these variables by enabling direct manipulation of the output of the program.

We explore diversity of program-based representations, focusing various paradigms of visual programming interfaces, from the imperative directed acyclic graphs (DAG) to the declarative Wang tiles, through more hybrid approaches. In all cases we study shape programs that evaluate at interactive rate, so that they fit in a creation process, and we push this by studying synergies of program-based representations with real time rendering pipelines.

We enable the use of direct manipulation methods on DAG output thanks to automated rewriting rules and a non-linear filtering of differential data. We help the creation of imperative shape programs by turning geometric selection into semantic queries and of declarative programs by proposing an interface-first editing scheme for authoring 3D content in Wang tiles. We extend tiling engines

to handle continuous tile parameters and arbitrary slot graphs, and to suggest new tiles to add to the set. We blend shape programs into the visual feedback loop by delegating tile content evaluation to the real-time rendering pipeline or exploiting the program's semantics to drive an impostor-based level-of-details system.

Overall, our series of contributions aims at leveraging program-based representations of shapes to make the process of authoring 3D digital scenes more of an artistic act and less of a technical task.

Résumé

Malgré la constante amélioration de la technique et du matériel informatique, permettant de manipuler du contenu numérique de plus en plus volumineux, la création de scènes virtuelles 3D reste une tâche complexe ; du fait notamment de la charge cognitive qu'elle impose aux artistes. Afin de fluidifier la création, des représentations d'ordre supérieur des formes 3D ont vu le jour : une forme est encodée en tant qu'elle est un programme qui génère sa géométrie. Cela rend possible une meilleure organisation de la charge cognitive lors de la création, mais possède néanmoins ses propres sources de friction. Nous étudions au cours de cette thèse les défis et opportunités induits par la représentation par programme des formes 3D, dans le contexte de la création de contenu numérique.

Nous cherchons à ce que l'interaction avec les formes reste autant que possible dans l'espace 3D, au lieu d'être une manipulation de symboles abstraits dans un espace de programmation. Il est question d'une part d'assister la création des programmes décrivant les formes, de permettre à l'artiste d'opérer dans l'espace 3D tout en assurant une bonne généralisation de ses actions lorsque les variables libres du programme sont modifiées, et d'autre part d'aider au contrôle de ces variables en permettant la manipulation directe de la géométrie générée par le programme.

Nous explorons la diversité de possibilités de représentation des formes par un programme, en nous focalisant sur différents paradigmes de programmation visuelle, allant des graphes orientés acycliques (DAG), impératifs, aux tuiles de Wang, déclaratives, en passant par des approches plus hybrides. Dans tous les cas, nous étudions des programmes de forme capables d'être évalués en temps interactif, de sorte qu'ils aient leur place dans un processus de création ; aussi étendons-nous notre étude aux synergies que ces représentations par programme peuvent établir avec les systèmes de rendu en temps réel.

Nous rendons possible l'utilisation de méthodes de manipulation directe sur la géométrie générée par DAG grâce à un jeu de règles de réécriture automatique et un filtre non linéaire de donnée différentielle. Nous aidons la création de programmes de forme impératifs en transformant des sélections d'éléments géométriques en des requêtes sémantiques, et la création de programmes déclaratifs en proposant un mode d'édition du contenu géométrique de tuiles de Wang centré sur les sections aux interfaces entre tuiles. Nous étendons les moteurs de pavage par tuiles pour prendre en compte des paramètres continus et suggérer automatiquement de nouvelles tuiles à ajouter. Nous intégrons les programmes de forme à la boucle de retour visuel en délégant l'évaluation du contenu des tuiles au système de rendu en temps-réel, et exploitons la sémantique du programme pour dériver un système de niveau de détails par imposteurs visuels.

En résumé, notre série de contributions vise à tirer parti des représentations par programme des formes pour faire du processus de création de scènes numérique 3D une tâche plus artistique et moins technique qu'elle ne l'est.

Contents

I Introduction **14**

I.1 The creation process: from intent to content 14
I.2 Representing shapes as programs 16
I.2.1 Shape programs help the creation process 17
I.2.1.1 Postponing artistic decision-taking 17
I.2.1.2 Non-destructive modeling in practice 18
I.2.2 Related types of parametric assets 18
I.2.3 Other creation workflows 20
I.2.4 Challenges specific to program-based representations of shapes 20
I.3 A taxonomy of shape programming paradigms 22
I.3.1 Terminology 22
I.3.2 Imperative Directed Acyclic Graphs 22
I.3.3 Declarative programs 24
I.3.4 Hybrid programming 25
I.3.5 Limits of our scope 27
I.3.5.1 About determinism 27
I.3.5.2 Real-time feedback 27
I.3.5.3 Non program-based higher-order representations 27
I.4 Outline 28
I.5 Contributions 28
I.6 Publications 29
I.6.1 Peer-reviewed papers 29
I.6.2 Released source code 30

II Related Work **31**

II.1 Shape programming in the wild . . . 31
II.1.1 Rigging . . . 32
II.1.1.1 Definition . . . 32
II.1.1.2 Assisted rigging . . . 33
II.1.1.3 Real-time evaluation . . . 34
II.1.2 Procedural Modeling . . . 35
II.1.2.1 Definition . . . 35
II.1.2.2 Use cases . . . 35
II.1.2.3 Constructive Solid Geometry . . . 38
II.1.2.4 Grammars-based modeling . . . 38
II.1.3 Inverse Procedural Modeling . . . 39
II.1.3.1 Symmetry detection . . . 39
II.1.3.2 Program synthesis . . . 40
II.1.3.3 Domain-specific methods . . . 40
II.1.4 Latent Space . . . 41
II.2 Interactive shape manipulation . . . 42
II.2.1 Geometry-level manipulation . . . 42
II.2.1.1 Direct mesh deformation . . . 42
II.2.1.2 Inverse Control . . . 43
II.2.2 Program-level manipulation . . . 43
II.2.2.1 Visual Programming . . . 43
II.2.2.2 DAG Rewriting . . . 44
II.2.2.3 Bidirectional Editing . . . 45
II.2.3 Authoring systems . . . 45
II.3 Optimization in hyper-parameter space . . . 46

III Imperative programming of shapes **47**
III.1 Introduction . . . 47
III.2 Automatic Synthesis of Semantic Selection Queries . . . 48
III.2.1 Problem setting . . . 49
III.2.2 Related work . . . 50
III.2.3 Overview . . . 50
III.2.4 Per-element trace recording . . . 51
III.2.4.1 Predicates . . . 51
III.2.4.2 DAG Amendment . . . 52
III.2.5 Domain Specific Language for Selection Query . . . 53
III.2.6 Query Synthesis . . . 54
III.2.6.1 Best program selection . . . 55
III.2.6.2 Program space exploration . . . 55
III.2.7 Results . . . 58
III.2.7.1 Experimentation . . . 58
III.2.7.2 Discussion . . . 60
III.2.8 Future Work . . . 61
III.2.8.1 Variants of the query language . . . 62

III.2.8.2 Integer expressions . . . 62
III.2.8.3 Variants of the synthesis algorithm . . . 63
III.3 Co-parameterization for the differentiation of parametric shape . 63
III.3.1 Introduction . . . 63
III.3.1.1 Problem Setting . . . 65
III.3.1.2 Related Work . . . 66
III.3.2 Co-parameterization . . . 67
III.3.2.1 Co-parameter definition . . . 67
III.3.2.2 Automatic DAG Amendment . . . 68
III.3.3 Results . . . 71
III.3.3.1 Implementation . . . 71
III.3.3.2 Limitations . . . 73
III.3.3.3 Future Work . . . 74
III.4 Jacobian Filtering: Applying Inverse Kinematics to Parametric Shapes . . . 76
III.4.1 Introduction . . . 78
III.4.1.1 Overview . . . 78
III.4.1.2 Sampling and differentiation . . . 78
III.4.2 Solving . . . 79
III.4.2.1 Inversion . . . 79
III.4.2.2 Jacobian buffer filtering . . . 81
III.4.3 Results . . . 84
III.4.3.1 Performances . . . 84
III.4.3.2 Ablation study . . . 85
III.4.4 Discussion . . . 86
III.4.4.1 Properties . . . 86
III.4.4.2 Limitations . . . 87
III.4.4.3 Future prospects . . . 87
III.4.5 Conclusion . . . 88

IV Tiles-based declarative programming of shapes 89
IV.1 Introduction . . . 89
IV.1.1 Constrained layout . . . 89
IV.1.2 Wang Tiles . . . 90
IV.1.2.1 Definition . . . 90
IV.1.2.2 Related Work . . . 91
IV.2 Tile-based geometric amplification . . . 93
IV.2.1 Problem Setting . . . 93
IV.2.1.1 Geometric amplification . . . 93
IV.2.1.2 Related Work . . . 94
IV.2.2 Method . . . 94
IV.2.2.1 Design workflow . . . 95
IV.2.2.2 Procedural mesostructure model . . . 96
IV.2.2.3 Tiling . . . 97

IV.2.2.4 Shell Mapping . . . 100
IV.2.3 Results . . . 102
IV.2.3.1 Experiment . . . 102
IV.2.3.2 Discussion . . . 103
IV.2.3.3 Future Work . . . 104
IV.3 Parametric tile content . . . 104
IV.3.1 Introduction . . . 104
IV.3.2 Method . . . 105
IV.3.2.1 Constraint propagation . . . 106
IV.3.2.2 Representation of a tile superposition . . . 109
IV.3.2.3 Sampling tile superposition . . . 109
IV.3.3 Results . . . 109
IV.3.3.1 Watershed generation . . . 110
IV.3.3.2 Discussion . . . 111

V Visual feedback of shape programs during authoring 112
V.1 Introduction . . . 112
V.1.1 A two-way integration of rendering and generation . . . 112
V.1.2 Related works and background . . . 113
V.2 Tile-based Mesostructure Rendering . . . 114
V.2.1 Method . . . 115
V.2.1.1 Render Pipeline . . . 115
V.2.1.2 Caching . . . 116
V.2.2 Results . . . 116
V.2.2.1 Performance . . . 116
V.2.2.2 Surface representation . . . 117
V.2.3 Discussion . . . 118
V.2.3.1 Properties . . . 118
V.2.3.2 Future work . . . 118
V.3 Multiscale Rendering of Dense Dynamic Stackings . . . 118
V.3.1 Introduction . . . 118
V.3.2 Pipeline overview . . . 120
V.3.3 Impostors for dense stackings . . . 121
V.3.3.1 General rendering pipeline . . . 121
V.3.3.2 Parametrization . . . 122
V.3.3.3 Sampling quasi-spherical impostors . . . 124
V.3.4 Model discrimination . . . 125
V.3.4.1 Impostors' validity range . . . 125
V.3.4.2 Dynamic grain splitting . . . 126
V.3.5 Occlusion Culling . . . 128
V.3.5.1 Grain-level culling . . . 128
V.3.5.2 Fragment-level culling . . . 130
V.3.6 Results . . . 131
V.3.7 Discussion . . . 135

V.3.7.1 Properties . . . 135
V.3.7.2 Limitations . . . 136
V.3.7.3 Future work . . . 136

VI Conclusion **138**
VI.1 Contributions . . . 138
VI.2 Future prospects . . . 139

Bibliography **142**

Appendices **166**
A Extra DAG Amendments . . . 166
B OpenMfx: Standardization of shape operators . . . 166
B.1 Technical approach . . . 166
B.2 Design Choices . . . 169
B.3 Future prospects . . . 171
C Proof A . . . 172
D Tile Rendering Statistics . . . 174
E Grain Rendering . . . 174
E.1 Impostor resolution . . . 174
E.2 Tables . . . 175

I Introduction

I.1 The creation process: from intent to content

Empowering creative people with the ability to author 3D content is one of the core goals of the field of Computer Graphics. Although hardware and techniques have increasingly improved over the years at handling heavy content (high amounts of polygons, complex lighting, etc.), digital 3D creation remains fairly complex. This is because the bottlenecks of a creation process do not only lie in compute power or memory limitations, they also include the saturation of the cognitive load on the designer. Addressing technical limitations consists in finding new algorithms to better solve existing problems, but reducing the cognitive load consists in identifying new problems, to reorganize the creation pipeline.

Unlike an engineering process, a creation process is not exclusively specified by a list of expected features. The only way to fully judge a creation is to face the result – or at least a relevant proxy[1] of it – and feel its impact. Hence, this inherently involves loops of trial and error: the designer tries a creation gesture, or a series of gestures – like brush strokes or mesh manipulation operations – then appreciates the result and often decides to backtrack and try something different. This dialog has a cost, especially when working as a team, where backtracking may require to ask multiple people to do retakes.

We intend to reduce the cost of this creative backtracking. Of course, this can partly be done by improving computing performance, because slow visual feedback or lagged interactions clearly increase the cost of the loops and interrupt the designer in their creative flow. But our key axis is different: we explore the possibility to postpone artistic decision taking at a later stage of the authoring plan, thus making interaction loops shorter. This is enabled by representations of shapes centered on

[1] We use the term *proxy* in the generic sens of an indirect substitute of a more accurate but less accessible goal, not specifically in the sens of a coarser geometry.

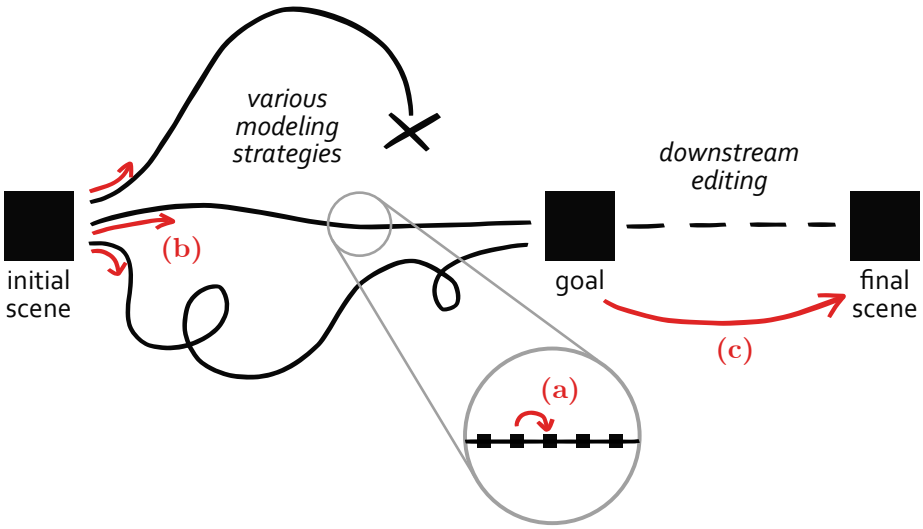


Figure I.1: *The process of creating a 3D scene or object requires the designer's anticipation at multiple levels: (a) local anticipation of the effect of each authoring gesture, (b) anticipation of the most appropriate modeling strategy and (c) anticipation of downstream use of their work.*

a *program*, namely a recipe leading to a shape, rather than the baked result itself.

> **NB** This culinary metaphor is widespread among 3D modeling tools. One says that a geometry is *baked* or *cooked* to mean that it is the result of an automated process, i.e., the output of a shape generating program. This term means that the geometry is stored only for caching purpose, it may be freed at any time as long as we still have the recipe. It is opposed to *editable* geometry, which does not have an underlying generative model.

Addressing sources of cognitive load

During the creation of 3D content, a designer's mind is busy *anticipating* the behavior of the tool and planning its modeling strategy, and is thus less dedicated to creative concerns (Figure I.1). This dialog between a digital tool and its user is studied by the field of Human-Computer Interaction (Beaudouin-Lafon, 2000), which serves as context to motivate our research.

Anticipation is needed locally, to choose a given authoring action. For instance, a tool that runs technically very fast but has a very chaotic response – a response that is hard to predict – is not an efficient tool. As a consequence, digital tools often adopt metaphors that mimic the physical world, because these are easier to anticipate for a human – who obviously is experiencing the physical world on a daily basis. And this is why we are interested in providing direct manipulation of the geometry in Chapter III.

Anticipation is needed more widely when splitting a goal into sub-targets corresponding to simple actions. This kind of gesture planning is called *workflow*. A good workflow avoids repetitive tasks, avoids stacking up too many nested sub-goals in the designer's memory, and allows for the serendipitous discovery of unplanned details without breaking the overall expected result. The creation tool is responsible for inducing a good workflow, hence multiple contributions of this thesis are presented as systems, rather than focusing solely on a particular algorithm. Nonetheless, workflow is also a matter of designer's craftsmanship, and actually the choice of the tool itself is part of the gesture planning.

Anticipation is needed even beyond the outcome of the designer's work, because 3D scenes are often produced in teams. For instance, when modeling a 3D mesh with the intent of animating it, the designer must keep in mind how their choice of polygon connectivity will affect deformation and light simulation applied later in the pipeline. In such a case, the designer is not producing a final creation but rather a proxy, so they must be able to evaluate whether it is good without being able to even try by themselves the later stages of the process. We focus on the use of parametric shapes as these proxies, so that they can be adapted later in the process rather than requiring one to fully anticipate downstream usage.

I.2 Representing shapes as programs

There are multiple ways of representing shapes, and in particular surfaces. Some are better suited for real-time rendering (triangle meshes), some are closer to raw acquisition devices (points clouds), some handle volumetric data (voxels), some ensure valid topology (implicit surfaces). In this thesis, we study *higher-order* representations, where the stored information is a logical program implementing a function $\mathcal{F}$ such that its output $\mathcal{F}()$ is one of the above-mentioned representations. By opposition, meshes, voxels, etc. are called *first-order* representations. For instance instead of storing a list of vertices and faces to describe a mesh, we store a program, which in turn generates a list of vertices and faces when executed. We call a program that outputs 3D geometry a *shape program*.

The first strength of this approach is to be able to turn internal constants of the program into free variables, provided as inputs to $\mathcal{F}$. For instance, if the program generates a 3-storey building by first generating a floor and then duplicating it, we can *expose* the number of duplicates as an input n and $\mathcal{F}$ then represents an n-storey building. We call these inputs *hyper-parameters* and note them $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_K)$. The function $\mathcal{F}$ thus becomes a *parametric shape* rather than a single fixed shape. An output $G = \mathcal{F}(\boldsymbol{\pi})$ is called an *instance* of the parametric shape.

Representing a shape as a program is also in general much more compact that a first-order model. In the worst case, the program is just an enumeration of geometric elements and is thus equivalent to a fixed shape. But, in general, shapes may feature a lot of symmetries that can get factorized into a compact program. A

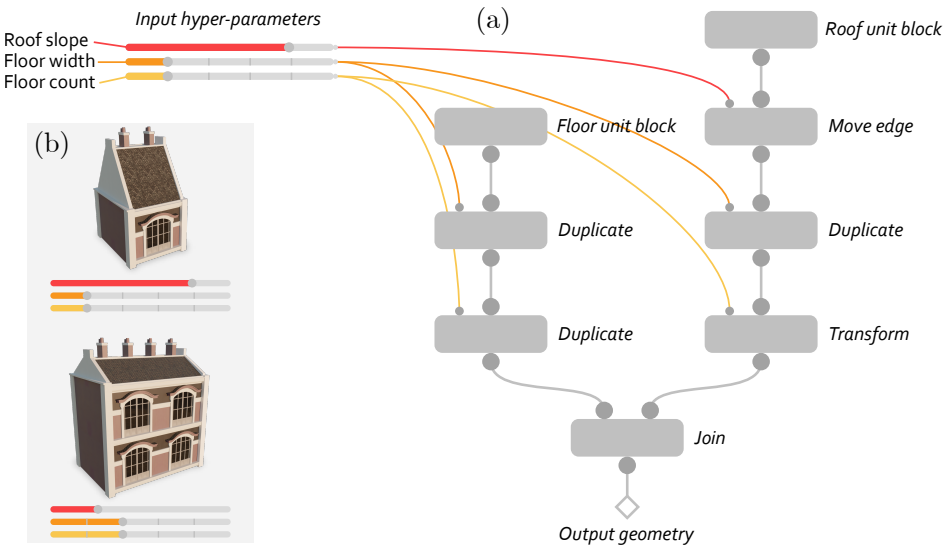


Figure I.2: *An example of a parametric shape and its program-based representation, displayed as an imperative DAG (a). We show two different instances of this shape, for different values of its hyper-parameters (b). Node inputs are typed: large discs are slot carrying geometric information, whereas small discs carry numbers. For the sake of legibility, the latter are generally omitted in our examples.*

program is also often resolution independent: the program "intersecting a sphere of radius 1 with a box of size $0.5 \times 0.5 \times 2$" remains a 12-word sentence, whether the representation of the sphere uses a high definition or not.

When the shape program is compact, and/or when it has been carefully authored by a designer, it contains information beyond its sole output: the program's structure tells a lot about the meaning of the shape and its parts. This semantic information is particularly required to ensure the consistency of the shape upon changes of the hyper-parameters, and reciprocally if a parametric shape behaves well, its program likely contains relevant semantic information. This observation is at the heart of our DAG amendment method presented in Section III.3.

I.2.1 Shape programs help the creation process

I.2.1.1 Postponing artistic decision-taking

When a designer crafts a regular static asset, they first define their intent, and then implement it through a series of authoring gestures. As they see their creation unfolding, they take many additional decisions on the fly, and burn them into the scene. For instance, when drawing a house, they eventually choose whether the slope of the roof is rather steep or flat and then build further, adding for instance tiles on the roof. Once done, if they realize that the slope does not feel right, backtracking almost means starting over, because all the actions happening after

this artistic decision depended on it.

On the contrary, if creating a parametric shape – represented as a program – the designer can leave all these decisions for a later stage by creating hyper-parameters that can still be tuned afterwards. Since we store a program, i.e., a series of instructions, we are able to automatically replay downstream gestures whenever the hyper-parameter defining the roof's slope changes.

The designer thus focuses separately on first the technical implementation of the shape program and second the detailed artistic choices. This reduces the need for the designer to go back and forth between high level intent planning, and lower level authoring actions, which is cognitively intensive. However it makes the process of building the initial shape more complex because one must account for all the possible values of the hyper-parameters, so in Section III.2 we explore one possibility to ease this process.

I.2.1.2 Non-destructive modeling in practice

The split between these two tasks, as well as the use of program-based representations for shapes to handle it, naturally emerged from the practice of 3D modeling in industrial contexts. Operations called *non-destructive* are available in all major 3D modeling tools, often called *modifiers* or *deformers*. They are effects applied as post processes, between the geometry actually manipulated by the designer and the output or display of the shape. They may be chained, and even reference the output geometry of other objects, for instance if the non-destructive operation is a Boolean difference. When used intensively, the combination of these modifiers constitute a graph of operations, that is in essence no less than a program.

Some tools pushed the use of non-destructive modeling one step closer to the idea of representing shapes as programs by basing their user interface mainly on the manipulation of a directed acyclic graph (DAG) of geometry processing operations, an interaction commonly described as *visual programming*. Some examples of this approach are *SideFX' Houdini*, the model graphs of *Adobe Substance 3D Designer* or the *Geometry Nodes* in *Blender*, plebiscited after the success of a third party extension called *Animation Nodes* (Figure I.3). Houdini has even been awarded an Oscar by the *Academy of Motion Picture Arts and Sciences* in 2018 precisely for its choice of a DAG-centric workflow, proving once again the practical relevance of this approach. Such imperative DAGs are the focus of our Chapter III.

I.2.2 Related types of parametric assets

Shapes are not the only kind of digital content that has been turned into parametric assets. The manipulation of procedural graphs of image filters is quite popular, either for texture design with tools such as *Adobe Substance Designer* or for video compositing like in *Foundry's Nuke*, *Natron* or *Autodesk Flame*. Parametric textures are of great value to help designers quickly explore variations of a texture, which

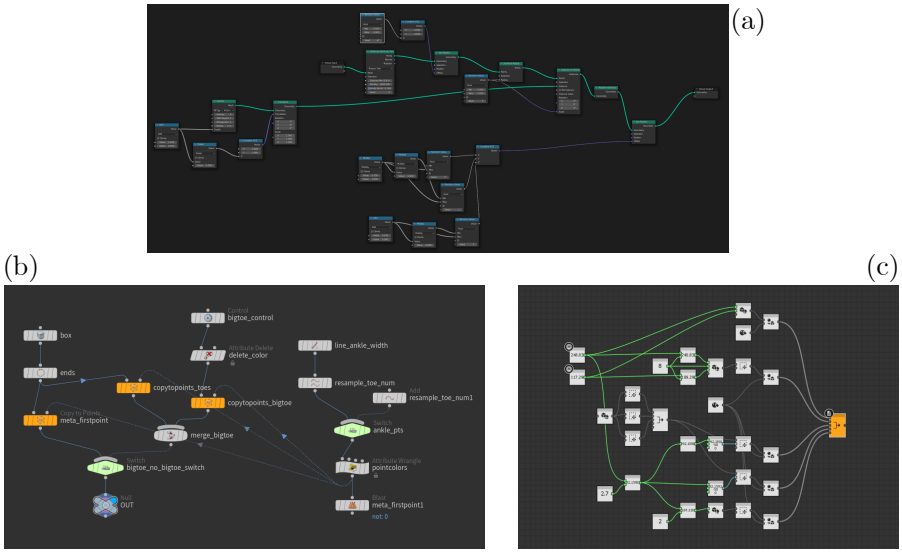


Figure I.3: *Examples of production-grade 3D modeling tools whose user interface is based on the manipulation of a DAG encoding a shape program: (a)* Blender*, (b)* Houdini *and (c)* Substance 3D Designer*.*

is not easy to do manually using an image editing program because a material is composed of multiple closely related maps (albedo, roughness, normal, etc.). For video compositing, the programmatic nature of the operations is important to ensure that effects are consistently applied to each frame of a video, something that would be very hard to achieve if the user would be manually drawing on each image individually. Actually even when the user interface of a compositing program is not explicitly showing a graph, e.g., in *Adobe After Effects*, the composition can still be thought as a program. Animation results from the progressive modification of the input hyper-parameters of the composition.

Sound processing is also a good client for parametric effects and program-based representations, especially in the context of live performances. From the very beginning of computer aided music, in the late 50s, with the *MUSIC* and *Csound* family of domain-specific languages, musical content was split into the description of parametric synthesizers on one hand and the specification of the temporal evolution of the so-defined parameters on another hand. These languages then inspired more visual approaches like *Max/MSP* and *PureData* (sometimes called patcher programming languages) which directly expose a visual DAG to the sound designer. Nonetheless text-based programming of music tracks is still in use, even for DJ-like live programming performances. This often relies on tools built around the *SuperCollider* programmable sound synthesizer (*TidalCycles*, *SonicPi*, etc.).

Patchers like *PureData* and *vvvv* were also extended to handle rhythm-based real-time graphic composition through visual programming, and a highly parametric

interface is actually found in all VJing tools, e.g., *Resolume* or *Smode*, or even *Notch* for live 3D effects. Indeed, a live performance set is typically prepared by building a set of dedicated parametric effects in advance and then on stage everything is achieved through parameter-tuning.

This thesis focuses on parametric 3D shapes used in the context of authoring objects or scenes for motion pictures and video games. This means we need shape programs to evaluate in interactive time, although true real-time is not as critical as in a live performance.

I.2.3 Other creation workflows

Parametric shapes are not the only way of mitigating the cost of creative loops. Tools as simple as layers, in 2D image manipulation applications like in *Adobe Photoshop*, are also meant for limiting destructive actions. Using different layers for sketching, coloring, lining, lighting, etc. also follows the principle of decoupling tasks.

Non-destructive workflows do not fit all situations. Virtual clay approaches (e.g., *Z-Brush*) are still relevant in some cases, for instance for concept arts at earliest stages of a production. This class of tools, including also virtual painting application, limits frictions in the creative process by pushing as far as possible the metaphor reproducing the feel of tangible tools like brushes and clay. Thus, the user can rely on their experience of the real world to anticipate the behavior of the tool. We use to some extent this metaphoric approach as a mean to make shape program manipulation more intuitive in Section III.4.

I.2.4 Challenges specific to program-based representations of shapes

Representing shapes as program has a lot of potential, but comes with a lot of challenges. Most of the difficulties are related to the fact that a program is meant to capture the semantic structure of a shape and that, with its hyper-parameter freed, a parametric shape is an object of very high dimension – too high to be visualized all at once.

Ensuring generalization Authoring a parametric shape is programming, and programming is a notably hard task, even if it is visual programming. Building a shape program is trickier than designing a single fixed shape because in order to evolve in a relevant way when the hyper-parameters later vary, the program must have a good power of generalization. The designer must at any time reason about all the possible variations of the shape and not just its current state.

For instance, when placing a jar on top of a table, the designer should not move it by a fixed amount. They should rather specify in the program that the jar moves vertically by a value equal to the height of the table, so that whenever this height

changes the jar remains on the table. The information actually stored in the model – the shape program – is then closer to the semantic assertion "being on top of the table" than it is a geometric piece of information. This is particularly visible in the case of declarative programming.

Assisted authoring of shape programs One of the challenges of shape programs is thus to assist the creation of the programs, make it feel as much as possible like crafting a single fixed shape, but without loosing the ability to generalize and produce relevant variations. This means being able to locally extract the semantic intent behind each edition of a shape, and providing atomic operations that operate at the scale of the whole program rather than on geometry.

However this assistance may introduce more constraint for the designer. Since finding the intent is an under-specified problem, we may need to integrate some domain-specific prior in the authoring tool. This raises the question of the artistic freedom of the designer: the modeled shape must be perceived as a creation of the artist, not a creation of the tool. A trade-off can consist in only suggesting several possible changes to the program, following different priors, and leaving the final word to the user. It is also important that the assistance does not completely prevent the user from manually editing the program. This two-way manipulation of a program and its output is sometimes referred to as *bidirectional programming*.

Shape manipulation Once written, the shape program still raises challenges. In the spirit of splitting the technical task of creating the program from the more art-oriented manipulation of the hyper-parameters, the latter should be as intuitive as possible for the user of the shape program – who is potentially not the same person as the program's designer. The main source of friction in this manipulation is that the degrees of freedom are hyper-parameters expressed in the space of the program, usually displayed as a list of sliders, but the intent of the user is better expressed in the 3D space, by directly manipulating the geometry output by the program. Interacting with raw hyper-parameters requires the user to anticipate the behavior of the parametric shape, which is one of the source of cognitive load that we intend to mitigate.

Integration with other programs A shape is processed by other tools than 3D modeling software; for instance it interacts with render engines or physic simulations. A straightforward way of ensuring the compatibility between shapes represented as programs and other tools is simply to evaluate the shape program and provide the resulting fixed geometry to, e.g., rendering algorithms. But the rendering pipeline could actually be adapted to benefit from the extra information embedded in the program. Real-time rendering in particular is important in authoring tools, since they must provide constant visual feedback to the designer.

I.3 A taxonomy of shape programming paradigms

I.3.1 Terminology

We make no particular distinction throughout this thesis between a shape representing a single *object*, often made of a single connected component, and a whole 3D *scene* containing many different pieces. We however associate a different meaning to the terms *shape* and *geometry*: the former is the concept of the object, or scene, whereas the latter is a formal subset G of $\mathbb{R}^3$, i.e., a set of 3D points. In particular, we can talk about a *parametric shape* as one entity, because however the hyper-parameters vary, we assume that the result still represents the same conceptual object. But we would not talk about a "parametric geometry" because for each value of the hyper-parameters, the output of the shape program is a different subset of $\mathbb{R}^3$, so a completely different geometry. Note that when dealing with a fixed shape, which has no possible variation, the notions of shape and geometry become equivalent. To summarize, usual first-order representations of shapes are meant to encode geometries, while higher-order representations (e.g., shape programs) encode a whole parametric shape.

> **NB** The term *parametric* is also used in geometry to qualify splines encoded as a function $u \in (0,1) \mapsto x \in \mathbb{R}^3$ and surfaces encoded as $(u,v) \mapsto x \in \mathbb{R}^3$. To avoid confusions with the parameters u and v, we stick to the term *hyper-parameter* to qualify the input $\boldsymbol{\pi}$ of our parametric shapes.

A *parametric shape* $\mathcal{F} : \boldsymbol{\pi} \mapsto G$ is a function. A *shape program* is a possible implementation of this function. In other terms, it is a *symbolic* representation of $\mathcal{F}$. These symbols may take multiple forms and follow different paradigms: imperative, listing a sequence of operations to perform, or declarative, stating a set of constraints to fulfill; deterministic or stochastic; visual or text-based.

I.3.2 Imperative Directed Acyclic Graphs

An imperative program is a sequence of effective actions, given in the order in which they must be performed to compute the output of $\mathcal{F}$ given its input $\boldsymbol{\pi}$. This is in a way the recipe of the shape. Imperative programming directly describes the control flow of the program and is thus close to the low level execution of the program – even though it also allows for higher-level abstractions. Although in other contexts a computer program is primarily seen as a text formatted with a specific syntax, in the case of shape programs we more naturally focus on Directed Acyclic Graphs (DAG). Manipulating a program as a DAG rather than as text may be called *visual programming* and is more common than text-based programming in the practice of shape programming.

Such a DAG is defined as a tuple (N, C) composed of a set N of *nodes* and a set C of oriented *connections*. A node represents a processing operation, and a connection

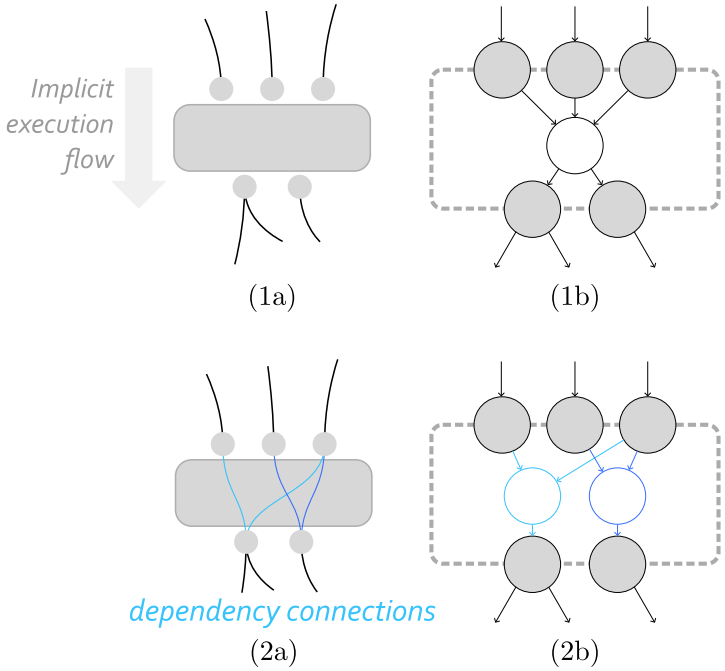
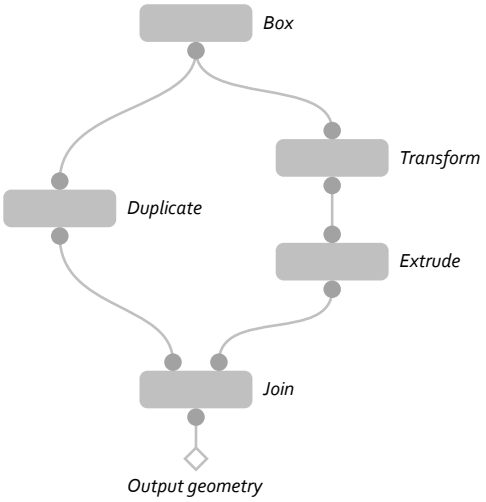


Figure I.4: *Our representation of nodes (1a) differs visually from the strict definition of a DAG, but is equivalent when encoding each slot as a node in the strict way (1b). In some contexts, dependency connections internal to our nodes are specified (2a) making encoding it as a slightly different strict DAG (2b).*

carries geometry from one operation to another. Each node has zero or more input slots, and one or more output slots, and a connection links one output slot to an input slot. An input slot may have at most one connection but an output slot can be connected to multiple inputs, because the geometry generated by an operation might be read by multiple other operations. Although the traditional mathematical definition of a DAG does not include this notion of slot, Figure I.4 shows that slots can be encoded in the pure definition, provided we can order the connections of internal nodes. In practical DAG editors, it is actually possible to nest a whole sub-graph inside a node, and to instantiate sub-graphs, providing an equivalent of function calls in a text-based program. Yet, this remains equivalent to a strict DAG when it comes to mathematical analysis.

Benefits of visual programming The original motivation for developing visual languages is likely the accessibility. When writing code as text, a programmer has to keep in mind constraints at two different conceptual levels: the syntax level, and the semantic level. Visual programming ensures the syntactic soundness of a program by-construction, leaving the developer with only the semantic part to think about. Of course, this part itself ranges over multiple levels of abstraction, from simple atomic operations to design pattern and software architecture, but



(a)

```
a = box()
b = duplicate(a)
c = transform(a)
d = extrude(c)
e = join(b, d)
return e
```

(b)

Figure I.5: *The same shape program, represented either as a visual DAG (a) or as an imperative text-based program (b). The former contains the extra information that operations `Duplicate` and `Transform` for instance may be executed in any order, or even in parallel, while the text-based representation required an arbitrary choice.*

at least syntax is no longer an issue. Furthermore, representing a program as a DAG is slightly more expressive than as a text in a purely imperative language: as illustrated in Figure I.5 and highlighted by Johnston et al. (2004), the DAG naturally provides information about what can be parallelized.

Seasoned programmers who developed habits with text-based programs easily feel frustrated when trying visual programming. It is in the best case slower to manipulate visual nodes than characters, but this is mostly because visual programming does not benefit from a base of productivity tools as large as what as been developed in the context of IDEs[2] for text manipulation.

I.3.3 Declarative programs

Explicit definition of the control flow of a program is not always the most convenient way to specify a program. Imperative programming requires to learn and explicitly implement a whole family of algorithms (although this effort can be shared through libraries of functions). But the designer of a shape often wants to express an intent that is not an algorithm but rather a set of constraints and requirements that the resulting shape must fulfill.

Declarative programming consists in declaring a series of facts and rules, and then letting a problem-agnostic engine solve the unknowns to provide a valid

[2]IDE: Integrated Development Environment

output. The engine can range from a solver of pure logic formulas to discrete or even continuous optimization. The former is the object of specific programming languages like *Prolog* or *Datalog*, while the later tends towards model fitting, where a template defined and constrained by the designer and/or by domain specific knowledge, and an optimizer finds the most relevant values for filling the template.

Since we are interested in visual programming, we focus on the framework of Wang tiles. A Wang tile is a square whose edges are marked with colors, and the only rule governing how tiles can be laid out is that only edges of matching colors are allowed to be in contact. Programming with Wang tiles consists in defining the set of slots that must be covered with tiles, as well as stating the edge colors for the set of tiles that can be used by the tiling engine. This constitutes a Turing-complete programming paradigm, so it is very expressive, while remaining fully visual. But this comes at the cost of being hard to solve in general.

I.3.4 Hybrid programming

In practice, the frontier between imperative and declarative paradigms is not as strict as presented above. For instance, Krs et al. (2020) proposes a powerful combination of imperative, declarative and example-based ways of modeling shapes. We illustrate the cohabitation of paradigms with a little case study of a video tutorial about shape programming where the author expresses their modeling strategy, which gives us insights about their reasoning. Figure I.6 shows steps extracted from this tutorial and we invite the reader to follow the video linked in its caption.

The first noticeable point is that, although they use a DAG-based imperative programming language, they actually recreate a shallow tile-based engine within this framework. The rules determining which tile belongs in a particular slot are very simple, purely contextual: they only consist in using the coordinates of the slot and the presence or absence of neighbors, and they don't involve advanced constraint solving. Yet this is one example among others showing that tile-based programming of shapes is an artist-approved creation tool.

The author mentions the term "semi-procedural modeling" to mean that the creation pipeline combines a shape program with static elements that have been manually authored, namely the tiles. This point is important to ensure that the artists can express themselves with enough freedom, that the result of the procedural modeling will wear their artistic signature. This ability to combine automated behaviors with hand-tailored elements is a strength of tile-based creation.

Furthermore, the artist deems some mensuration of the tile's content as "flexible". They do not really use it in practice because their framework does not make this easy enough, but they thus mean that the tile's content features hyper-parameters which may be tuned to create variations. Interestingly, this means that within the declarative framework of Wang tiles, the content of tiles can be parametric shapes

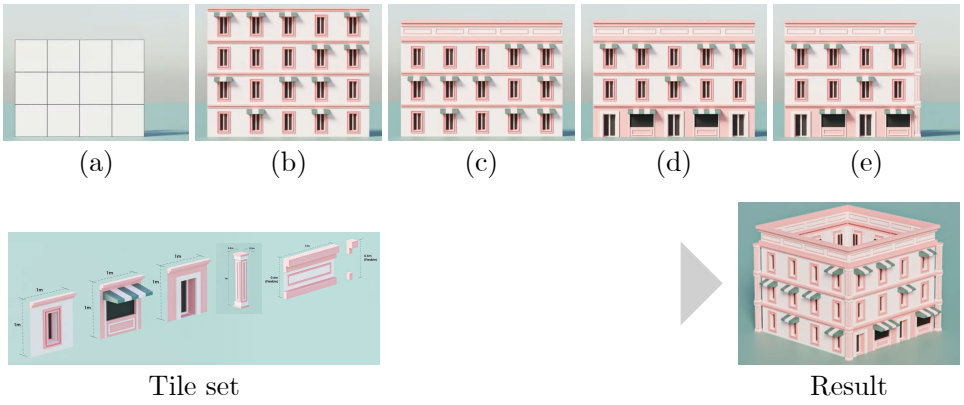


Figure I.6: *Steps (a) to (e) are captured from the explanation of the procedural modeling strategy adopted by the designer to create the result shape. Although this is in practice implemented using an imperative DAG (Blender's* Geometry Nodes*), this approach is close to tile-based modeling, where step (a) describes a graph of slots where tiles of geometry are instanced following specific adjacency rules such as placing shops and doorways if there is no slot bellow.* Courtesy of Chong 3D (https://www.youtube.com/watch?v=-rexNuTap44).

described by an imperative program.

One challenge raised by this idea is that some neighboring constraints may entangle the hyper-parameter values of one tile with those of another tile. For instance here the "flexible" value is the height of the tiles used for the top-most row: although it may vary from one building to another, it must be the same for two neighbor tiles. We investigate this idea of making tiles themselves parametric shapes in Section IV.3.

Lastly, on a slightly different topic, this practical example of shape programming shows the importance of the *semantic selection* scheme that we will explore in Section III.2. A consequent part of the tutorial focuses on the problem of selecting the top-most row of tile slots. They first use a threshold on the vertical coordinate of the slot, but as they highlight, hardcoding the threshold fails at generalizing to a change in the number of floors. They then switch to a procedurally defined threshold that depends on the floor count, thus representing the selection as a program. This selection program behaves well in all cases, but the process for defining this selection query is far from being as simple as manually selecting the intended row. We intend in Section III.2 to simplify this process thanks to a query synthesis method able to generate a semantic selection given a manual selection.

I.3.5 Limits of our scope

I.3.5.1 About determinism

Whether it is declarative or imperative, a shape program – like any other program – may be either deterministic or stochastic. A deterministic program always outputs the same geometry when it is given the same input, whereas a non-deterministic program features random behaviors. This non-determinism can come from the intentional use of a random generator in the program, or in the case of a declarative program it may be caused by the use of random algorithms in the core engine, when multiple solution can fulfill the constraints.

Although randomness can contribute a lot to the visual diversity of a program-generated shape, it requires more anticipation effort from the user, or arouse frustration if they want to roll back to a previous result that pleased them better. It is hence common to turn non-deterministic programs into deterministic ones by exposing the random *seed* of an internal pseudo-random generator as an input hyper-parameter. This way, the user can change unrelated hyper-parameters without triggering the re-sampling of all (pseudo-)random values. Hence we focus on deterministic programs.

I.3.5.2 Real-time feedback

We focus on programs that evaluate at interactive rates, so that they fit in a fluent creation loop. As a consequence simulation and heavy procedural generation, although being programs which output geometry, are out of the scope of this thesis.

I.3.5.3 Non program-based higher-order representations

Representing a higher order shape $\mathcal{F}$ – which is a function – through its implementation as a program is the most natural option, and the one we focus on in this thesis, but it is not the only one.

Besides the symbolic approach, a function can be represented by its graph. In our case, the graph of $\mathcal{F} : \boldsymbol{\pi} \mapsto G$ is a subset of $\Pi \times \mathbb{R}^3$. A key difference though is that we very often need to query slices (at constant $\boldsymbol{\pi}$), which is not something usual representation have been optimized for. There is thus no obvious representation for efficiently encoding the graph of a parametric shape $\mathcal{F}$.

Another class of non-symbolic representations of parametric shapes is the case of learnt representations (a.k.a. data-based representations), and in particular neural networks.

I.4 Outline

The remainder of this thesis is organized as follows. Chapter II presents previous work related to representing shapes as programs. In particular, we see that although the wording may vary, many assets typically found in modeling work are in essence parametric shapes.

Chapter III explores possibilities to bring the interaction with DAG-based imperative parametric shapes back in their output 3D space, rather than in program space as it is usually the case. We address this issue both at design time, assisting the creation of such shape programs, and at manipulation time, by providing inverse control of the input hyper-parameters.

Since imperative DAGs, with explicit control flow, are not the only programming paradigm, we study in Chapter IV another class of shapes, based on constraint programming. We focus in particular on tile-based modeling, a versatile framework for discrete constraint programming of shapes, and hybrid it with the previous chapter by turning tiles into imperative shape programs themselves.

Since our motivation to study shape programs is their positive impact on the creative loop, we see in Chapter V how to integrate them more deeply in the pipeline for real-time visual feedback critical to the efficiency of an interactive tool. This coupling enables us to either defer the evaluation of some parts or the shape program or anticipate rendering within this program, and thus increase the performance of rendering.

Finally, Chapter VI concludes this thesis by opening to new prospects that seeing shapes as program may offer.

I.5 Contributions

Our main contributions are summarized in this section. A first part unifies the manipulation and automatic processing of imperative DAG-based shape programs:

- An automatic DAG amendment mechanism for defining shape differentiation with respect to the hyper-parameters even when the DAG outputs geometry of varying connectivity.
- A non-linear kernel for filtering jacobians when applying inverse kinematics on procedural geometry with high-frequency spatial variations, rather than on a coarse control structure.
- A program synthesis method for turning selections of geometric elements that were hand-picked on a particular instance into semantic selection queries generalizing consistently when the program is altered.

Some other contributions focus on declarative tile-based programming, illustrated

on a design workflow for authoring mesostructure along surfaces:

- A bottom-up approach to the creation of the geometric content for Wang tile-based modeling, bringing the interfaces between tiles as the primary area of edition and ensuring continuity by construction.
- A tile suggestion mechanism to assist the creation of a set of Wang tiles, mitigating the difficulty of tiling being NP-hard.
- The hybridization of tile-based modeling and imperative procedural models thanks to a tiling engine solving for per-tile continuous hyper-parameters.

At last come contributions on the integration of shape programs whose output features a lot of self-similarity with real-time visual feedback:

- A compact tile-based mesostructure model where geometry evaluation and mapping are delegated to the GPU, enabling real-time user feedback and delivering hundreds of millions of polygons per-second on standard hardware.
- An efficient rendering pipeline for dense stackings of similar procedural objects, leveraging their quasi-spherical geometry to enable advanced culling at both object and fragment level.

I.6 Publications

I.6.1 Peer-reviewed papers

A large proportion of the contributions discussed in this thesis has been presented as standalone papers in international scientific journals, and thus validated by our peers:

- Élie Michel and Tamy Boubekeur. 2020. *Real Time Multiscale Rendering of Dense Dynamic Stackings*. In *Computer Graphics Forum* (Proceedings of Pacific Graphics), volume 7-39, pages 169–179. (É. Michel & Boubekeur, 2020b)
- Élie Michel and Tamy Boubekeur. 2021. *DAG Amendment for Inverse Control of Parametric Shapes*. In *ACM Transactions on Graphics* (Proceedings of SIGGRAPH), volume 4-40, pages 173:1–173:14. (É. Michel & Boubekeur, 2021a) Awarded **best scientific production** for the STIC prize (by Paris-Saclay doctorate schools).

We also presented two posters:

- Élie Michel and Tamy Boubekeur. 2020. *Real time multi-scale sand rendering*. Poster in *Symposium on Interactive 3D Graphics and Games (I3D)*. (É. Michel & Boubekeur, 2020c) Awarded **best poster**.

- Élie Michel. 2021. *OpenMfx: An API for cross-software non-destructible mesh effects*. Poster in *SIGGRAPH*. (É. Michel, 2021)

And we presented each year our ongoing work in the local French conference of Computer Graphics:

- Élie Michel and Tamy Boubekeur. 2019. *Rendu de sable multi-échelle en temps réel*. In proceedings of *Journées Françaises d'Informatique Graphique et de Réalité Virtuelle (JFIGRV)*. (É. Michel & Boubekeur, 2019)
- Élie Michel and Tamy Boubekeur. 2020. *Formes paramétriques différentiables*. In proceedings of *Journées Françaises d'Informatique Graphique (JFIG)*. (É. Michel & Boubekeur, 2020a)
- Élie Michel and Tamy Boubekeur. 2021. *Synthèse par pavage de mésostructure surfacique*. In proceedings of *Journées Françaises d'Informatique Graphique (JFIG)*. (É. Michel & Boubekeur, 2021b) Awarded **second best paper**.

I.6.2 Released source code

Throughout the making of this thesis, multiple prototype have been released as open source programs. This section briefly points the reader to the chapters they relate to.

DagAmendment An add-on for Blender enabling the direct manipulation of parametric shapes built using non-destructive tools such as modifiers and constraints. It regroups the contributions of sections III.3 and III.4.

MesoGen Standing for *Mesostructure Generator*, it is a modeling tool for authoring complex mesostructure along the surface of quad meshes, following the tile-based approach of Section IV.2. Its real-time visual feedback is ensured by the method from Section V.2.

GrainViewer A standalone viewer focusing on dense stackings of pseudo-spherical items which demonstrates the multi-scale real-time rendering pipeline presented in Section V.3.

II

Related Work

Our choice of studying shapes that are represented as programs is grounded in the observation that, despite a variety of wording to designate them, such shapes are already present in diverse contexts. The notion of *program* is indeed wide enough to be the common denominator of many representations. It might sometimes even feel too generic to be informative, so in each chapter we will particularize our analysis, nevertheless we draw up in Section II.1 a landscape of program-based representations to stress out the potential of our overall approach.

Furthermore, we have identified that the strength of shape programs lies in the shape manipulation workflow they enable. Section II.2 introduces interactive mesh deformation methods and then focus on authoring tools that involve parametric shapes and their programmatic nature. Both geometry-level and program-level manipulation of shape can be combined together in bidirectional authoring systems.

Lastly, we review in Section II.3 techniques that pragmatically explore the hyperparameter space of a parametric shape, looking for a solution optimal with respect to some problem-specific criterion. This setting is typical from Computer-Aided Design, a field where shapes are often represented as programs and the correct variation is selected based on physical constraints rather than artistic choices. Although this might sound different from our area of study, Section III.4 treats user's input gestures, during shape manipulation, as a particular case of problem-specific criterion to automatically optimize for.

II.1 Shape programming in the wild

Many objects manipulated in a shape modeling pipeline can be thought as parametric shapes, and thus naturally represented as programs. Parametric shapes are indeed a natural way to represent 3D objects in a space of lower dimension

and higher meaningfulness than giving direct access to e.g., raw vertex positions. Section II.1.1 highlights that *rigging* is the de facto tool used in practice to build a large part of parametric shapes.

More generally, programming is the most effective framework for representing complex and multifarious intents; Section II.1.2 presents shape programming idioms developed in the context of Procedural Modeling.

A program carries semantic information about the shape it generates. Inversely, semantic information can be used to build a program given its output. This extraction is the challenge addressed by Inverse Procedural Modeling techniques presented in Section II.1.3.

Lastly, we review in Section II.1.4 how the product of Machine Learning tools such as auto-encoders can be thought as parametric shapes. They are however parametric shapes whose representation is not a symbolic program but rather a numerical model.

II.1.1 Rigging

II.1.1.1 Definition

Rigging is the act of transforming a given static geometry into a parametric shape. This step comes after the static modeling and prior to the animation of a shape, and its main purpose is to restrain the possible deformations of the geometry, to ensure that whoever animates it may use only meaningful poses.

The most common use of rigged deformation is character animation. The rig ensure for instance that an elbow cannot be bent in the wrong direction, or that a foot cannot suddenly become twice as big as it usually is. The terminology used by the rigging literature is thus inspired by its application to characters, although the notion of rigging is actually more general. Any model meant to be articulated can be rigged, including for instance mechanical structures.

Rigging is a three-way process. At first one defines a coarse kinematic structure called *skeleton* or *armature*. It is a graph of *joints* – 3D points with an orientation and a scale – connected to each other by *bones*. Joints are equipped with various constraints and expressions that reduce their degrees of freedom and expose them as higher-level semantic values to the animator. For instance, the direction of the eyes of a character is better exposed as a position to look at rather than an angle of rotation of the eyeballs. Secondly, one attaches the actual surface of the object to these bones, a process commonly refer to as the *skinning*. And finally, building a rig includes the creation of controllers. Controllers are visual handles whose position, orientation and scale drive the joints; they enable spatial interaction with the abstract degrees of freedom of the skeleton.

The result of rigging is a parametric shape. The semantic degrees of freedom of

the skeleton are what we call hyper-parameters, and the deformed character or object is the output of this shape. The rig itself is in a way a program. It is at first glance a declarative program since building a skeleton consists in defining constraints. But the rig engine does not intend to solve these constraints using an advanced optimization scheme, because it is important that the program executes in real time for the animators to do their job. Instead, it treats the constraints as imperative operations and figures out in which order they must be executed.

The rig engine thus compiles a declarative program into an imperative one, which is by the way usually exposed to the designer as a DAG. Of course this has limitations, it fails as soon as there are dependency loops in the constraints, because topological sorting is then not possible. It covers nonetheless a wide range of common use cases, and more advanced scenarios can be addressed by manually interacting with the DAG.

II.1.1.2 Assisted rigging

Some of the difficulties of shape programming that we identified previously have been studied in the context of rigging. We focus here on assisting the rigging process, and shape manipulation will be discussed later on, in Section II.2.

We distinguish the *kinematic parameters*, which are the raw degrees of freedom of the joints of a coarse skeleton, or control cage, and the higher-level *hyper-parameters* $\boldsymbol{\pi}$ on which the animator has control. Wording varies among papers, for instance Capell et al. (2005) calls them respectively *pose parameters* α and *abstract parameters* β. Hyper-parameter space is also sometimes called embedding space, rig space (Hahn et al., 2012), design space (Talton et al., 2009) or animation space (Merry et al., 2006).

Kinematic parameters can be estimated automatically using geometric analysis for skeleton extraction, especially in the case of organic objects. This can be based for instance on path finding in the medial surface of a voxelized object (Tsao & Fu, 1984; Wade & Parent, 2000), or using the Reeb graph of the geodesic distance to a single user input point (Lazarus & Verroust, 1999; Hilaga et al., 2001; Tierny et al., 2006; Pascucci et al., 2007; Aujay et al., 2007). Some extraction methods may produce loops (Au et al., 2008), which is not ideal for rigging but still enables animation.

On the other hand, determining semantic hyper-parameters requires a domain specific prior. One possibility is to fit an existing hand-made rig space to a new input geometry (Baran & Popović, 2007; H. Li et al., 2010; Ali-Hamadi et al., 2013), typically for motion re-targeting (Avril et al., 2016). This works for shapes that are common to many different scenes, such as human bodies for instance, and enable the use of animation databases such as *Adobe Mixamo*. For less common shapes, e.g., imaginary creatures, Miller et al. (2010) developed *Frankenrigs*, which composes parts of different example rigs. For more examples Rumman and Fratarcangeli

(2016) surveys auto-skinning methods and includes discussions about skeleton extraction.

The other way of injecting prior knowledge is through the use of Machine Learning. It has been applied to most common use cases of rigging like human bodies (Anguelov et al., 2005; Loper et al., 2015; L. Liu et al., 2019; Osman et al., 2020), human faces (Blanz & Vetter, 1999; T. Li et al., 2017; Vesdapunt et al., 2020; Song et al., 2020) or even generic shapes (Z. Xu et al., 2019). For instance L. Liu et al. (2019) automatically skin characters using Graph Convolution Networks, even when the skeleton varies. Holden et al. (2015) learns how to place semantic manipulators to reach a given kinematic pose. Generally a subset of the hyper-parameters gives the morphological identity of the character and the remainder is related to posing and animation.

> **NB** Softwares like lucky3d's *AutoRigPro*[1] shows the large variety of tools that can be used to help the tedious process of rigging. In practice this kind of assisted tools seems to be preferred by artists over fully automated approaches.

II.1.1.3 Real-time evaluation

Animators work with time, they compose a temporal signal, and the human eye is very sensitive to tiny variations of delays and synchronization. The ability for a rig to evaluate in real-time is thus critical. However, a production-grade rig can easily be composed of several hundreds of operations, in order to account for numerous details and cases of non-linear deformation.

Watt et al. (2012) reports how the animation studio *Dreamworks* schedules the evaluation of the nodes of a rig in order to exploit multi-threading. Similarly, *Pixar* developed a rig execution engine called *Presto* (ElKoura & Studios, 2013) and *Disney Animation* also addressed these challenges (Lin et al., 2015). A Siggraph 2014 course on multi-threading (Watt et al., 2014) surveys these solutions centered on computer engineering. This can be combined with on-GPU rigging/skinning that is used in the context of video games (Tarini, 2017), as well as memory management methods to reduce the I/O complexity (Marchal, 2018).

When optimizing the evaluation of the rig is not enough, for instance because there are multiple characters moving together, a solution is to replace the rig with a faster but equivalent model of deformation. For instance Bailey et al. train a deep learning model to mimic the behavior of complex rigs but at interactive rates (Bailey et al., 2018, 2020). Their work intervenes between the creation of the rig and its use by the animator, similarly to our DAG amendment in Section III.3 (see Figure III.20).

[1] https://blendermarket.com/products/auto-rig-pro

II.1.2 Procedural Modeling

II.1.2.1 Definition

Repeated patterns and systematic behavior in the designer's authoring gestures and observations are common during 3D modeling. The automation of these creation rules is called procedural modeling, and its outcome naturally takes the form of a shape program. It enables the production of heavy content that would take too much time to craft manually, like immense landscapes, cities, diverse crowds, complex trees, etc. As highlighted by R. Smelik et al. (2010), procedural modeling papers study, for a particular application, the tension between artistic freedom i.e., the versatility of the procedure, and the degree of automation and speed of creation.

Our intent on the other hand is to abstract the process of developing procedural models. We need the shape to be represented as a program, but try to remain agnostic in the specific domain it was developed for. In the case of imperative procedural models, Schinko et al. (2011) and before them Havemann's thesis (2005) stated a similar intent. In particular the Chapter 5 of the later, about Generative Modeling Language, describes a program-based representation of shapes.

The boundaries of procedural modeling are somewhat fuzzy. On one side, it fades into heavier simulation methods. Although technically a rigid body simulation is a procedure that automates creation (specifically the animation), it is not considered as a case of procedural modeling. A simulation algorithm reproduces a physical phenomenon, but does not embed any artistic bias. It may however be a part of a procedural modeling system. On its other side, we start calling *parametric shape* a procedural model when the program evaluates in interactive time. But when the procedure has a chaotic behavior i.e., when its output is hard to predict, subject to a lot of pseudo-randomness, the term procedural modeling remains preferred.

II.1.2.2 Use cases

Procedural generation is used to model virtually any kind of shape in practice, but its use cases that end up in the academic literature are centered around a few particularly challenging tasks. We introduce some of them here and for a more exhaustive overview we refer the reader to dedicated surveys like R. M. Smelik et al. (2014) and Krispel et al. (2014), or the state-of-the-art section of previous thesis centered on procedural modeling (Emilien, 2014).

Terrain generation The fractal beauty of natural landscapes has long been a source of both fascination and challenge for artists and researchers. Earliest work consisted in crafting fractal noise models generating mountain-looking elevation maps (Prusinkiewicz & Hammel, 1993; Ebert et al., 1994), whose hyper-parameters were amplitude, lacunarity, recursion depth.

These models were missing important visual features caused by weathering, so shallow simulation algorithms for thermal and hydraulic erosion were later introduced and refined (Beneš & Forsbach, 2002; Mei et al., 2007; Št'ava et al., 2008). They are usually constrained to layered heightmap representation of terrains (which is well described by Cordonnier (2018)), although some also use more general fluid simulation techniques, e.g., smoothed-particle hydrodynamics (SPH) (Krištof et al., 2009).

As mentioned above, this kind of time-consuming simulation-based algorithm no longer fits our conceptual framework of parametric shapes, supposed to evaluate in interactive time. So other approaches were developed to give water streams a primary role in landscape shaping from the bottom up (Kelley et al., 1988; Belhadj & Audibert, 2005), leading to large-scale consistent watersheds (Génevaux et al., 2013, 2015; Peytavie et al., 2019).

Tile-based methods, on which we focus in Chapter IV, have also been used for terrain generation, as well as other combinatorial approaches (Maung & Crawfis, 2015). It is particularly used in the context of video games (Stalberg, 2018), where tiling also provide gameplay-related features, e.g., path-finding (Scurti & Verbrugge, 2018; Sandhu et al., 2019). We use terrain generation as an example of application of our parametric tiling method in Section IV.3.

A more complete view of procedural modeling methods can be found in surveys from R. M. Smelik et al. (2009) and Galin et al. (2019).

Data-based generation Another edge of procedural modeling is the use of existing data, either as a prior, a set of examples or a bank of samples to query from. For instance, H. Zhou et al. (2007) introduced a powerful way to create elevation maps using existing samples that comply with a complex user input. The rise of machine learning techniques, which started around the same time, lead more people to look into this kind of data-based approach to terrain generation (É. Guérin et al., 2017; E. Guérin et al., 2022).

In a way, data-based procedures may feel like a degenerate cases of procedural generation, because they have to either embed their whole dataset or use a compact representation of its data that is hardly human-interpretable (e.g., neural networks), but when the latter rely on automatic differentiation tools, it can actually fit the approach we use in Section III.4. Search-based procedural generation is surveyed by Togelius et al. (2011).

Trees and plants Foliage is a particular feature of landscape that is a research field in itself. Here again, the challenge lies in the highly fractal nature of shapes, leading this time to the use of L-Systems (Lindenmayer, 1968). Initially introduced in the field of botany and biology, this grammar-based method has quickly been reused for procedural plant generation in computer graphics (Mech, 1997;

Prusinkiewicz, 1999; Prusinkiewicz et al., 2000). Conceptual simple yet quite versatile, L-Systems enable the interactive manipulation of their input hyper-parameters (e.g., average branching angle), as opposed to methods leaning more towards simulation (de Reffye et al., 1988). Interactive authoring eventually became the focus of tree modeling techniques, leading to production tools like the one of Lintermann and Deussen (1999) or the award-winning *SpeedTree*, which is centered on a declarative programming model. Historical plant generation methods are surveyed in Deussen and Lintermann (2005).

Buildings and cities Less natural than mountains and forests, cityscapes are nonetheless a source of fascination as well. The combination of seemingly systematic rules and huge amounts of varying content makes them a good fit for procedural generation, both at the scale of whole cities and individual buildings, and even for room layout. As for plants, grammar-based programming is often used to represent buildings, especially following Müller and Wonka's work (Wonka et al., 2003; Müller et al., 2006). Shape grammars are also used for facades, and it is noticeable that Haegler et al. (2010) integrate such a program-based representation in a real-time rendering pipeline, similarly to what we discuss in Chapter V.

At the scale of a building, procedural modeling is used by architects themselves, as largely reviewed by the *Parametric Architecture* publishing platform (*Parametric Architecture*, 2016) and supported by node-based modeling tools like Rhino's *Grasshopper* (Rutten, 2007).

There is here again a wide variety of approaches to city generation. Some use tiling (Gaisbauer et al., 2019; Stålberg, 2020), some use data-based approaches (Aliaga et al., 2008), some target villages, interacting with terrain and natural landmarks (Emilien et al., 2012), maybe focusing only on roads (Galin et al., 2010) or bridges (Patow, 2011). For further details, city generation is surveyed by G. Kelly and McCabe (2006). In our context, the boundaries of parametric shapes usually stop at the scale of buildings or bridges, but generating a whole city on the fly is possible as well (Steinberger et al., 2014).

Going down to the level of interior layout, procedural models generally conform to a more declarative paradigm, like Le Roux et al. (2001) which clearly state their contribution as a layout solver. This solver might be data-based (Merrell et al., 2010) and this apply to both room layout and furniture arrangement (Germer & Schwarz, 2009; Merrell et al., 2011).

Garment and fabric It is also a common target of procedural modeling, especially when modeling up to the scale of individual yarns (Yuksel et al., 2012). It usually includes some sort of simulation (for relaxation), might feature tile-based elements (Leaf et al., 2018), and is challenging to include in real-time viewport (K. Wu & Yuksel, 2017). We will present other types of intricate mesostructures in Section IV.2.

II.1.2.3 Constructive Solid Geometry

A Constructive Solid Geometry (CSG) tree (Requicha, 1977) is a representation of a shape where internal nodes represent boolean operations (union, difference, intersection) and leaf nodes are primitive shapes (spheres, cylinders, boxes, etc.). Primitives are generally parameterized with a few dimensions. It is a simple yet powerful program-based representation, which is at the heart of Computer Aided Design (CAD) modeling.

A CSG tree does not depend on an underlying first-order representation of the geometry. Although it can be converted into a mesh (Laidlaw et al., 1986), this is not needed for rendering. A CSG tree may be directly raytraced (Roth, 1982), evaluated as a signed distance field for sphere tracing (Hart, 1996), or drawn using screen-space CSG rendering (Goldfeather et al., 1986; Kirsch & Döllner, 2004; Zanni et al., 2018). The latter are well suited for interactive visual feedback, and yet an example of synergy between a shape program and a real-time rendering pipeline (Chapter V).

Nevertheless, when extending CSG trees with other operations than booleans, which is a natural path towards more generic imperative DAG-based shapes, one needs efficient polygonal CSG like QuickCSG (Douze et al., 2017).

II.1.2.4 Grammars-based modeling

We mentioned above multiple cases of grammar-based programming of shapes. These approaches are inspired by the formal grammars, originally developed by Chomsky (1956) in theoretical linguistics and largely used in the programming language literature.

Such a program is expressed as a set of production rules operating on string of abstract symbols. For instance, a rule $X \rightarrow aXb$ states that any occurrence of the string X will be replaced with the string aXb. Starting from an initial symbol, the recursive application of production rules creates a word. In an L-System (Lindenmayer, 1968), this word is interpreted as drawing instructions. In a shape grammar Stiny and Gips (1971), each symbol has a spatial embedding (e.g; a bounding box) and terminals (symbols that are allowed in the end word) correspond to a primitive shape.

A grammar-based program is imperative, but also often non-deterministic, because in general multiple rules may be applied at the same location in the word. As suggested in Section I.3.5.1, we can consider that using a pseudo-random generator with a fixed seed is enough to see it as a deterministic program. Nevertheless, recursive stochastic makes the behavior of ambiguous grammars very chaotic, so Talton et al. (2011) proposes a Metropolis-Hasting (Metropolis et al., 1953; Hastings, 1970) algorithm to enable a more direct control of the output. This purely discrete inverse problem is at the opposite end of the range of direct manipulation methods;

in Section III.3 we rather focus on continuous hyper-parameters.

Shape grammars can be encoded into imperative DAGs, as illustrated by Patow (2012) and others (Silva et al., 2013, 2015). Hence, we will not particularly focus on shape grammars in our formalism. For a deeper review of grammar-based procedural methods, the reader may refer to Lienhard's thesis (2017).

II.1.3 Inverse Procedural Modeling

Each specific domain of application requires its own procedural models, but developing such a model is a very technical and time consuming task. Inverse procedural modeling (Aliaga et al., 2016) aims at automating, or at least assisting, this challenge. It transforms a static geometry, for instance a 3D scan, into a semantic program. Inverse and forward procedural modeling are very related because, as we stressed already, the edges of procedural modeling are blurry: a procedural system may integrate inverse tools to handle advanced user input such as sketches (Nishida et al., 2016).

An inverse procedural modeling pipeline is a combination of geometrical analysis and program synthesis. The analysis part identifying symmetries that could be factorized, both locally and at the scale of larger structures, and program synthesis assembles this information into a program that can provide semantic control and produce variants of the original shape. Both steps might be helped with domain-specific priors.

II.1.3.1 Symmetry detection

The local geometrical analysis of a shape for inverse procedural modeling can be based on the detection of self-similarities or local symmetries, which are reviewed by Mitra et al. (2012). It might also be based on primitive fitting (Kaiser et al., 2019; Lê et al., 2021). The latter uses a slight prior that fosters manufactured objects, whereas symmetry detection is purely intrinsic, thus well suited for organic shapes. However, the choice of which of these approaches to use also depends on the synthesis method one targets.

Detecting self-similarities can also come as a by-product of compression algorithms. For instance, compressing voxel data leads to a DAG-based representation where similar parts are factorized at all scales (Kämpe et al., 2013). As a matter of fact, in information theory a lower bound of the size of a compressed signal is the Kolmogorov complexity, defined as the minimal size of a program that generates the signal.

Symmetries can be extracted at multiple levels and consolidated into higher-level blocks (Kalojanov et al., 2016). This is in a way a first step towards the construction of a program.

II.1.3.2 Program synthesis

Figuring out the declarative rules, or imperative process, behind the structure identified by symmetry detection methods is a particular case of Program Synthesis (Winston, 1970; Summers, 1977). This branch of the programming language literature addresses the problem of automating the tedious task of programming itself. Naturally, this is an overly difficult problem in general, so it usually focus on particular cases. Here we review computer graphics research that lean towards program synthesis, although sometimes not presenting it as such. For a more general introduction to program synthesis, see Solar-Lezama (2018).

A good example of program synthesis for inverse procedural modeling are *CAD decompilation* (Nandi et al., 2018) and *InverseCSG* (Du et al., 2018). The latter first uses simple primitive extraction using a RANSAC-based method, before leveraging program sketching (Solar-Lezama, 2008) to efficiently explore the space of CSG programs whose leaves are the fitted primitives. They thus decouple the combinatorial problem of primitive assembly from the continuous optimization of the dimensions of primitives.

Automatic construction of CSG trees has also been addressed using reinforcement learning (Sharma et al., 2018, 2020). Actually, reinforcement learning is more generally used to help program synthesis (Johnson et al., 2017; Bunel et al., 2018), and sometimes it is even the other way around (Yang et al., 2021). Reinforcement learning is used by Ganin et al. (2018) to auto-encode images in program-based representations, instead of the usual latent vector. It can also be used to learn shape grammars for instance (Teboul et al., 2011; Martinovic & Van Gool, 2013).

Part-based modeling (Ritchie et al., 2018) learns the rules for laying out existing building blocks. Sometimes the rule inference is only used as a sub step to directly re-synthesize content, without providing an explicit program-based representation (H. Liu et al., 2015). On the contrary, ShapeAssembly (Jones et al., 2020) learns to synthesize and interpolate shape programs from a larger database of examples. ShapeMOD (Jones et al., 2021) refines this by improving the factorization, carrying self-similarity detection in the space of programs rather than geometry.

We come back on program synthesis in Section III.2, with the more specific intent of generating database queries – where the database is a mesh, in our case.

II.1.3.3 Domain-specific methods

Inverse procedural modeling techniques can hardly be fully agnostic in the content they handle, and some of them utilize strong priors: for trees (Šťava et al., 2010; Stava et al., 2014), for facades (F. Wu et al., 2014), for buildings (Demir et al., 2016), for cities (Vanegas et al., 2012), etc. Model-fitting can be thought as an extreme case of domain-specific inverse procedural methods where existing procedural methods are adapted to the input geometry. Zhao, Luan, and Bala (2016) follows

this approach for yarn-level inverse modeling of clothes, and many automatic rigging methods like *Frankenrigs* (Miller et al., 2010) are in essence model-fitting.

When the prior is strong enough, the input may differ from 3D geometry. For instance, Nishida et al. (2018) uses an image as input to extract procedural building models, following a long history of image based modeling (Debevec et al., 1996). In a similar spirit, the same authors had proposed a powerful interactive workflow for authoring buildings from sketches (Nishida et al., 2016). It becomes difficult to consider this as inverse procedural modeling per se, but it sure follows the same intent of assisting the creation of advanced parametric models from simple data.

There has been recently a growing attention for the inverse procedural modeling of materials (Hu et al., 2019; Shi et al., 2020; Hu et al., 2022). As for *InverseCSG*, these methods generally disentangles the discrete exploration of possible program structures from the optimization of its continuous parameters. Possible structures are often sampled from a bank of predefined programs, making this a case of model-fitting. When targeting 3D rendering, materials and geometry are very related, so there is room for bringing these work to cooperate with inverse procedural geometry.

II.1.4 Latent Space

Our tour of fields whose output is shape program ends with the edge case of latent-space encoding. In the context of machine learning, it is common to learn a low dimensional abstract space, called *latent space* or *embedding space*, to represent a set of shapes. The *decoder* model, which generates a geometry given a particular point of this latent space, is an example of parametric shape: the latent vector is the set of hyper-parameters. A decoder may operate on any first-order representation: voxels (Girdhar et al., 2016), surface patches (Groueix et al., 2018), signed distance field (Z. Chen & Zhang, 2019; Eisenberger et al., 2021), vector displacement (Eisenberger et al., 2021).

This approach is nevertheless particularly interesting in the case of complex but very common shapes, such as human bodies. SMPL (Loper et al., 2015) and STAR (Osman et al., 2020) learn skinning weights and blend shapes from actual data, and produces a model that fits the usual rigging pipeline. Mahmood et al. (2019) uses these model to unify human representations from multiple datasets. FLAME (T. Li et al., 2017) follows a similar path to build a parametric human face.

Latent spaces often have more dimensions than a human designer can handle though (typically a few hundreds). Dimensions can be sorted by importance or reduced to a sub-manifold (Chiu et al., 2020; Abdrashitov et al., 2020) to reduce the number of dimensions, bringing latent space even closer to our conception of hyper-parameter.

Similarly to our notion of hyper-parameter, it intends to model semantic axes of

variation of the shape. Latent space is fully continuous, which provides shape interpolation, and decoders are automatically differentiable, which fits well into learning pipelines. However, the decoder provides no direct access to the logic linking these semantic dimensions to the end geometry. We thus have parametric shapes that are not represented as what we call a *program*.

As a consequence, learned embeddings shifted to the goal of shape decomposition into high level parts (Paschalidou et al., 2021). This can replace symmetry detection in inverse procedural modeling, extracting semantic building blocks with strong priors, like volumetric primitives (Tulsiani et al., 2017), superquadrics (Paschalidou et al., 2019), structured implicit functions (Genova et al., 2019, 2020), convex shapes (Deng et al., 2020) or even partly interpretable space partitioning trees (Z. Chen et al., 2020). The self-similarity analysis is in a way shared across a whole dataset.

II.2 Interactive shape manipulation

We study program-based representations of shapes in the context of content authoring. We hence put our goal in perspective with other means of manipulating shapes on one hand, and program on another hand.

II.2.1 Geometry-level manipulation

II.2.1.1 Direct mesh deformation

Deforming *raw* geometries that are not the output of an underlying parametric shape requires extra prior knowledge. Some methods try to maximize rigidity (Igarashi et al., 2005; Levi & Gotsman, 2015), sometimes based on examples (Sumner et al., 2005; Wampler, 2016). Linear variational methods (Botsch & Sorkine, 2008) deform the input by solving a linear system capturing the intrinsic properties of the mesh, enriched with direct control constraints coming in the form of vertex handles. Non-linear methods (Botsch et al., 2006; Sorkine & Alexa, 2007) further develop this concept, to better preserve volumes and cope with large handle motions. Alternatively, linear blend skinning (Baran & Popović, 2007; Jacobson et al., 2011) offers a scalable framework where no system is solved at run time, and the bulk of the shape analysis yielding the handles influence is located at the initial per-vertex weight computation.

When manipulating a parametric shape, we want to provide such direct control capabilities but our case differs significantly, as our a priori is the space of possible embeddings a shape can undergo through variations of its hyper-parameters. This is somehow an extreme case of structure-aware shape processing (Mitra et al., 2014), although such method usually couples the user deformation (change of the hyper-parameters) with the extraction of symmetries (X. Wu et al., 2014; Kurz et al., 2014), of coarse control structures from the geometric analysis of one (Gal et

al., 2009; Bokeloh et al., 2012) or many similar shapes (Gadelha et al., 2020).

Improving interaction with parametric shapes has been explored by T. Kelly et al. (2015) who automatically places the hyper-parameter controllers in the 3D view, but the controllers themselves must have been hand-designed first.

II.2.1.2 Inverse Control

Manipulating the output of a shape program can be seen as a case of Inverse Kinematics (IK). IK takes its source in robotics (Saab et al., 2013) and has been extensively studied for skeletal animation (Aristidou et al., 2018). Although their announced scope is often limited to trees of rigid transforms, the methods proposed in the IK literature may be applied to more complex mesh deformations like human face posing (Lewis & Anjyo, 2010). The main requirement is indeed only to access the jacobian matrix of the action of hyper-parameters onto a point of the mesh. With this jacobian at hand, methods have been developed to solve robustly the inverse problem (Deo & Walker, 1992) and account for boundaries of the hyper-parameters (Baerlocher & Boulic, 1998; Raunhardt & Boulic, 2007). Inverse kinematics can be done online or off-line (especially for motion planning in robotics). Since we are designing an interactive tool, we are interested in online solutions.

A major issue, which we address in Section III.3, is the difficulty to define a reliable way to measure this jacobian matrix when the connectivity of the geometry changes and so vertex indices cannot be used to identify points. Some hyper-parameters are not even continuous. In this case, IK can only apply on proxies, or even not apply at all and strategies like sampling-based exploratory modeling Talton et al. (2009) can be adopted.

II.2.2 Program-level manipulation

In program-level manipulation, the designer edits the shape by altering its program rather than operating on baked first-order geometry. There exists multiple modalities of program-level manipulation, depending on the paradigm of shape programming, the domain of application and the type of user input.

We deal with the same kind of tension than for geometry-level manipulation: we want to apply more constraint during manipulation, to reduce the boilerplate, but without paying a price in generality and ambiguity.

II.2.2.1 Visual Programming

The most straightforward solution to program-level manipulation is raw edition of a text-based program. But more advanced tools tend to leverage visual programming in order to mitigate syntactic friction (a lot of valid texts are not

valid programs) and high level of abstraction. The strengths and variety of visual programming are well introduced by Burnett (1999), and the problem of the cognitive load of text-based interaction was largely studied in the Human Computer Interaction (HCI) literature when advocating for direct manipulation approaches (Shneiderman, 1981; Hutchins et al., 1985).

Visual programming can take any form, from early executable charts (Ellis et al., 1969) to data-flow visual programming (Hils, 1992; Johnston et al., 2004), through block-based programming, like *Boxer* (diSessa & Abelson, 1986), *Scratch* (Resnick et al., 2009) or domain-specific languages derived from *Blockly* (Marron et al., 2012). Visual programming has been applied early to computer graphics (Haeberli, 1988) in order to reach more easily a creative audience. There used to be reviews of visual programming in general (Myers, 1990) but the ubiquity of the concept led research to focus on more restricted scopes.

The HCI literature also coined the notion of *End-User Programming* (EUP) to mean that a large part of computer programming is performed by users who just need to express their intention but do not aim at becoming professional programmers ever (Lieberman et al., 2006; Myers et al., 2006). Authoring shape programs clearly is a case of EUP since we expect the expertise of designers to be more about art than about computer science.

> **NB** EUP is closely related to program synthesis, and in particular to *Programming by Example* (Myers, 1986) or *by Demonstration* (Lieberman, 2001). The latter is applied to shape programming by Girard (2001) to assist the creation of CAD models.

II.2.2.2 DAG Rewriting

It is sometimes not needed to start a shape program from scratch; one can rather start from an example – or a previous work – and incrementally edit it. This has been particularly studied for the edition of shape grammars (Barroso et al., 2013), for instance to perform program-space shape interpolation (Lienhard et al., 2017). Lipp et al. (2019) transforms edits applied by the user on a particular instance of a procedural shape into edits of the original split grammar.

For more generic imperative shape programming languages, Jones et al. (2021) achieves automatic factorization of shape programs using machine learning, and Mathur et al. (2020) assists the creation of generative programs by transforming hand selections into semantic queries. The latter highlights precisely what makes interactive editing of shape programs challenging: spatial interaction is more intuitive, but always ambiguous. This is the main motivation of our Chapter III, and in particular their program synthesis approach is closely related to Section III.2.

II.2.2.3 Bidirectional Editing

The relevance of *bidirectional editing* is well illustrated by Gruber et al. (2020), which shows that in a character modeling workflow the artist may want to alternatively modify semantic hyper-parameters (the shape program's input) or edit spatial features (in the program's output).

The notion of bidirectional programming as theorized in the programming languages literature is largely reviewed in Foster's thesis (Foster et al., 2007; Foster, 2009). The term has been ported to shape programs by Chugh and Hempel when designing *Sketch-n-Sketch* (Chugh, 2016) and its follow-ups (Chugh et al., 2016; Hempel & Chugh, 2016; Mayer et al., 2018; Hempel et al., 2019). This tool enables the modification of procedural 2D vector graphics either by manipulating code or through direct spatial interaction with the output.

In its simplest form, bidirectional programming enables the modification of the scalar constants of the program, and it can already be challenging to apply to 3D shapes, as presented by our Section III.3 or by Cascaval et al. (2022) and Gaillard et al. (2022).

More advanced bidirectional programming intends to enable changes in the structure of the program. But as this is often way too ambiguous, an interesting trade-off is to help the designer locate "good edit locations" in the program and let the designer do the changes by themselves. This can be applied to web pages (X. Wang et al., 2012), shape grammars (Lipp et al., 2019), or CAD programs (Mathur et al., 2020).

The translation of the concept of bidirectional programming to a declarative paradigms a bit unclear, but for instance smart snapping tools (Schulz et al., 2014; Ciolfi Felice et al., 2016) are in a way an adaptation to constrained-based programming.

II.2.3 Authoring systems

> *The main goal [...] is to develop a system whose representation and processing facilities correspond to and assist the mental processes that occur during creative thought.*
>
> — David C. Smith, in *PYGMALION* (1975)

Like Smith, in this thesis we consider the task of content authoring as a whole, from the expression of an intent and its interpretation by the tool to the visual feedback (Chapter V). Although their work is now dated, Smith presented a system that was already relying on visual programming (Iconic Programming) and program synthesis (a sort of Programming by Demonstration). Their implementation was limited to specific engineering applications, but their philosophical introductory discussion about the relationship between creation and computers resonates with

artistic applications.

We consider program-based representations of shapes because they are a concept broad enough to cover many cases of authoring systems. Even the earliest interactive graphic creation systems like *Sketchpad* (Sutherland, 1964) included features akin to constrained shape programming and instancing.

Nevertheless, the means of human computer interaction can take so many forms (Beaudouin-Lafon et al., 2021) that some systems hardly fit in this framework. For instance, *AttribIt* (Chaudhuri et al., 2013) proposes a verbal design workflow, where natural language is used to tune the semantic attributes of the edited object's parts. *Facade* (Debevec et al., 1996) is a modeling tool which combines a coarse program-based representation of shapes as well as photograph, from which program's inputs and additional geometric details are extracted. Another example is the whole body of work that focuses on sketch-based modeling e.g., for landscapes (Ponjou Tasse et al., 2014), plants (Longay et al., 2012), or for shape retrieval (Shin & Igarashi, 2007; Eitz et al., 2012; Nishida et al., 2016).

II.3 Optimization in hyper-parameter space

We presented parametric shapes and their program-based representations as object manipulated by human designers. However they can be used in contexts where the space of input hyper-parameters is automatically explored by an optimizer. Typically, the user designs a parametric shape that represents a space of acceptable shapes, and a solver looks for the most suitable output according to some additional criterion.

The optimized criterion can be provided by a mechanical simulation, for instance to ensure equilibrium (Whiting et al., 2009) or to express physical phenomena in the rig space, so that they cooperate with hand-tuned animation (Hahn et al., 2012).

The constraint can also consist in matching photographs, for human pose estimation (Zhang et al., 2020), or to determine the hyper-parameters of a shape program (Debevec et al., 1996).

Lastly, one can combine user interaction and automated optimization to help the exploration of design space (Shugrina et al., 2015; Schulz et al., 2018), or to match a user gesture, as in any inverse kinematic scenario. For a goal similar to ours Section III.4.3, Gaillard et al. (2022) automatically explores the hyper-parameter space to identify multiple directions matching a spatial user edit and cluster related solutions. For machine learned models, or more generally any differentiable model, optimization can be used to move in the input latent space by dragging output vertices (Umetani, 2017).

III

Imperative programming of shapes

III.1 Introduction

This first core chapter focuses on imperative DAGs, which are the most common visual language used for shape programming. Designing and tuning a shape represented as a DAG is canonically done in program space, by manipulating symbolic nodes and abstract value sliders. We intend to bring some of the interaction back into the 3D space, which is more intuitive to interact with.

Although we tried to remain to some extent agnostic in the first-order representation of the geometry produced by the shape program, this chapter mainly treats the case of 3D meshes. Some experiments with signed distance fields are also presented to show the potential of generalization of our approach.

The first two sections present techniques based on *DAG amendments*. They are shallow DAG rewriting mechanisms applied on the fly to augment the output of the DAG with extra information. In a sense, a DAG amendment is a meta-modifier, referring to Blender's meaning of "modifier", namely a procedural mesh post processing effect.

In Section III.2 we amend the DAG to record a trace in each element that we use to assist the creation of a DAG in the specific case of procedural selection of geometry. Section III.3 presents a DAG amendment that enables a differentiation of parametric shapes, telling how the input hyper-parameters affect each element of the geometry. Section III.4 uses this differential information to provide a mean to directly manipulate the output of a DAG, bridging the gap with usual Inverse Kinematics setups. A less research-oriented outcome of our work on DAG is presented in Appendix B, which addresses the more practical problem of formaliz-

ing a common programming interface for DAG operators, in order to harmonize representations of DAGs across modeling toolkits and enable interoperability.

III.2 Automatic Synthesis of Semantic Selection Queries

There are two ways of editing a shape resulting from the evaluation of a DAG. One of them consists in applying an operation to the output geometry as if it was any static geometry: deforming an area, extruding a face, etc. This is equivalent to appending a new node at the very end of the initial generation DAG. The other possibility consists in altering more deeply the DAG, changing arbitrary internal nodes.

The first option can easily be automated: each time the user interacts in the 3D space with the shape, a new node is created. This is what a lot of tools actually do to keep track of the edition history. The operations do not need to be aware of the programmatic nature of the shape, all they process is the previous output, so any usual mesh processing technique can be applied.

However, such an history-based approach to DAG creation results in programs that are very unlikely to return a meaningful output when one alters past operations. There are two main reasons that explain this lack of generalization: **(i)** this approach produces a degenerate graph, which misses factorization and, more importantly, **(ii)** the values of node parameters are too specific to the very instance of the parametric shape that was being visualized when appending the node. For instance if the designer translates some geometry by 5 units along the X axis, the value 5 is recorded as-is in the history-based DAG, rather than recording the process that led the designer to chose this value.

We have no direct access to this decision process, since it runs mainly in the designer's mind, but we can try to guess. For real or integer valued parameters, this problem relates to alignment detection methods, e.g., *StickyLines* (Ciolfi Felice et al., 2016).

But there is a specific type of node parameter that requires a different approach: *selection parameters*. Many operations take as input a selection parameter, that states which area of the geometry must be affected. It is a set of vertices, edges and/or faces, and the most straightforward way to represent it is as a list of indices. For instance an extrusion operation is usually not meant to extrude all faces, so it takes a list of face indices as input. But if upstream operations are modified and produce a mesh with a different number of faces or a different connectivity, the index of the faces that the designer intends to extrude is very likely to change.

We propose to replace the index-based representation of selections with a program-based representation. We call this program a *selection query*, and we synthesize

this query from an example of input mesh and output selection set.

Contributions

- A method that turns a hand-picked selection of geometric elements into a selection query i.e., a program that output a list of vertices, edges or faces. This program can be applied to other geometries resulting from the same or a similar upstream DAG of operations.
- A procedure that automatically augments the DAG to provide per-element history features to help the query synthesizer.

III.2.1 Problem setting

Let $\mathcal{F} : \boldsymbol{\pi} \mapsto G$ be a parametric shape whose output $G = \mathcal{F}(\boldsymbol{\pi}) = (V, E, F)$ is a 3D mesh containing vertices V, edges E and faces F. We focus on vertices in this section, but the same applies to edges and faces. Let $\mathcal{S}$ be a *selection* of vertices, namely a subset of V. Let us now consider another instance $G' = \mathcal{F}(\boldsymbol{\pi}')$, and its vertices V'. Our problem is to build a selection $\mathcal{S}' \subset V'$ that is *semantically equivalent* to $\mathcal{S}$ (Figure III.1).

To define the notion of semantic equivalence, we first define the *program-based equivalence*. For a program $Q : v \mapsto \{true, false\}$, we deem $\mathcal{S}$ and $\mathcal{S}'$ as Q-equivalent if $\mathcal{S} = \{v \in V \mid Q(v) = true\}$ and $\mathcal{S}' = \{v \in V' \mid Q(v) = true\}$. This means that Q encodes the process that leads to selecting some vertices and not some others, and that this process is the same for $\mathcal{S}$ and $\mathcal{S}'$.

We say that $\mathcal{S}$ and $\mathcal{S}'$ are semantically equivalent if there exist a *well formed* program Q such that $\mathcal{S}$ and $\mathcal{S}'$ are Q-equivalent. A well formed program is a program that could have written by a human and fully represents their intent. Although this definition seems as ill-posed as just claiming that two point sets are *semantically equivalent*, it turns our specific case into the fundamental problem addressed by the *Program Synthesis* literature.

Our problem thus becomes to synthesize a program Q that behaves well on the examples of V. For each vertex $v \in V$ we want $Q(v) = true$ if $v \in \mathcal{S}$ and $Q(v) = false$ otherwise. This set of examples enables us to draw from techniques of *Programming by Example*.

This problem is well illustrated in our introductory example (Section I.3.4, Figure I.6), when the designer spends a lot of time ensuring that they split correctly the top row from the other of slots: G is a grid and $\mathcal{S}$ is its first row, and if G' is a different grid, $\mathcal{S}'$ must still be the first row. The best representation of the selection is the sentence "the first row", rather than point indices. Although it is conceptually a very simple query, actually encoding it into the program-based representation critically affects the creation flow.

III.2.2 Related work

Creation workflows commonly rely on an alternation of selection and operation (Nishida et al., 2016), so the importance of a good selection process not restricted to program-based representations (Guy et al., 2014). Nevertheless, the problem of generalization to unseen instances of a shape program brings in specific challenges. It is not sufficient to transfer the selection attribute across instances using shape correspondence (van Kaick et al., 2011) because **(i)** it does not take into account the information embedded in the program and **(ii)** does not enable symbolic manipulation and audit of the selection rules.

The CAD literature identified the problem of automatically *naming* entities produced by a shape program (X. Chen & Hoffmann, 1995; Capoyleas et al., 1996; Kripac, 1997; Agbodan et al., 2000), which is the point addressed by our trace mechanism, but generally without proposing a creation workflow around. In the specific case of shape grammars, Lipp et al. (2019) transform spatial changes into program-space edits. Mathur et al. (2020) concurrently made the same observation than ours, and developed a similar approach, leveraging program synthesis and the notion of bidirectional programming. They use a different query language, which is more focus on CAD applications and geometric patterns, and a different synthesis algorithm. A key difference is our use of a trace-based entity naming.

Our algorithm is inspired by bottom-up recursive program synthesis (Albarghouthi et al., 2013), and we maintain explicit copies of individual partial programs – as opposed to Version Spaces for instance (Lau et al., 2003). Program synthesis easily focuses on database queries (C. Wang et al., 2017) as it usually implies an more constrained program-space to explore. Also, instead of using a pre-existing query language, we tune ours in order to ease the synthesis.

The idea of recording traces has already been used, for bidirectional programming of 2D vector graphics (Chugh, 2016), or as a mean to augment the input of subsequent stages that rely on machine learning (Yang et al., 2022), even post rendering (e.g., denoising). In our case, it is used to augment the input of a more symbolic learning process, namely program synthesis.

III.2.3 Overview

We intend to synthesize a program Q whose input is a vertex v and which outputs a boolean $b \in \{true, false\}$, given a set $\{(v_i, b_i)\}$ of examples. We first present in Section III.2.4 the input features that Q receives to describe v, which we call the *trace* of v. This is all Q may use to decide whether v should be selected. Secondly we define the Domain Specific Language (DSL) that Q is implemented with (Section III.2.5). Lastly, we detail the program synthesis algorithm that we use to explore the program space of this DSL and find the right Q (Section III.2.6).

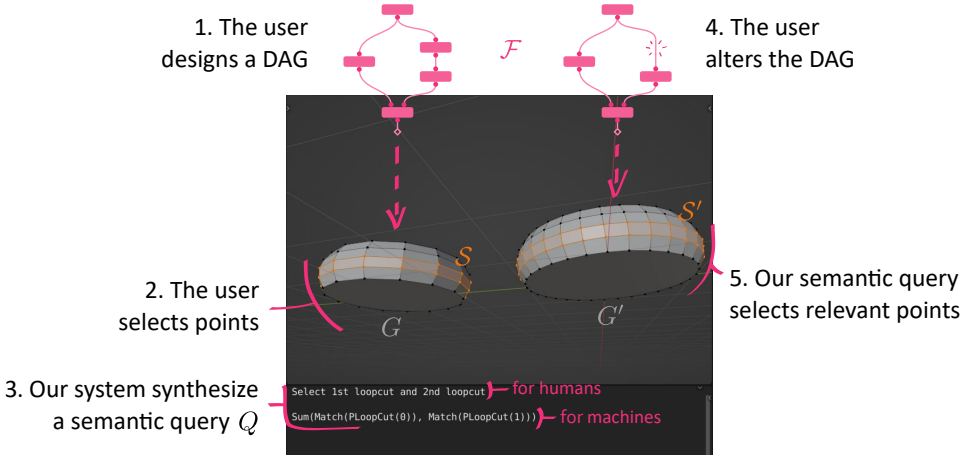


Figure III.1: *Our query synthesis takes place in the following workflow: the designer defines $\mathcal{S}$ by hand selecting geometry in one instance of a DAG-generated shape (2.), then our methods infers a program Q which is equivalent to the selection (3.) such that when the DAG changes (4.) our program Q produces a* semantically equivalent *selection $\mathcal{S}'$ in the new geometry (5.).*

III.2.4 Per-element trace recording

When we say that the target program Q takes as input a vertex v, we must define more exactly which data it receives. Except for really simple cases, the index of v is not enough, so we provide other input features to Q. These features are **(i)** geometric information and **(ii)** historic information.

Geometric information consists in both spatial data, such as the position of a vertex or the normal of a face, as well as connectivity data e.g., a number of neighbors. In this report though, we focus mainly on historic information, leaving geometric properties to Mathur et al. (2020) and future work.

III.2.4.1 Predicates

Historic information tells which modeling operations affected the elements of geometry, and what role the element was playing during the operation. Each element e.g., face f of the geometry is thus labeled with a *trace* $\mathcal{T}_f$, namely a list of *predicates* of the form *Operator* (Role). For instance, in an *extrusion* operation, the face that was extruded plays a different role than the new side faces created by the extrusion. This role is represented as a list of parameters.

Symbolic role Predicate parameters can be abstract symbols; for instance, in the previous example, extrusion faces are labeled with the predicates respectively *Extrude* (FRONT) and *Extrude* (SIDE).

Integer role Predicate parameters may also be integers; for instance to differentiate duplicates in a *repetition* operation or to loops in a *loop cut* operation. Integer parameters are also used for predicates representing primitive generators: a grid generator initializes the trace of each vertex (i, j) with a predicate $Grid(i, j)$.

In a selection query, integers can be either treated as if they were abstract symbols, or be involved in integer expressions, using modulo to carry on dashed selection such as "select one point every three points".

III.2.4.2 DAG Amendment

In order to compute the trace $\mathcal{T}$ of each geometric element, we must alter the nodes of the DAG, so that they append their associated predicate to the geometry they process. In practice we do not need to fully rewrite DAG nodes, but rather to wrap them into trace wrangling operations. This non invasive approach, which we call *DAG amendment*, enables ones to integrate easily into existing DAG-based frameworks, and we will use it again in Section III.3.

There are two points to consider to assign a trace $\mathcal{T}_e$ to an element e generated by a node n, namely **(i)** the role that e played in n, and **(ii)** the previous trace of e, to which the new predicate must be appended.

Role The first point must be treated for each node type individually (each possible operation), but is usually quite straightforward. For instance for the extrusion operation, when implementing our method in *Houdini* we use the groups that the extrusion node can create for the side and the front geometry, and in *Blender* we can rely on vertex indices to characterize the front geometry. We call P_e^n the predicate produced by the node n for its output element e.

Parent trace Difficulties arise when it comes to finding the history of e prior to n. We need to related elements of the output mesh of a node to the elements of its input mesh (or meshes). Here we make the assumption that, either by wrapping the node or in its core behavior, we can find a weighted set $\{(e'_1, w_1), \ldots, (e'_k, w_K)\}$ of elements e'_k of the input meshes with weights $w_k \in (0, 1)$ which sum to 1.

When $K = 1$, we define the trace $\mathcal{T}_e$ of e as the input trace $\mathcal{T}_{e'_1}$ appended with P_e^n. When $K = 0$, we are unable to relate the new element to any previous one, so we initialize $\mathcal{T}_e$ to $[P_e^n]$. When $K > 1$, we compute the *longest common subsequence* LCS of the $\mathcal{T}_{e'_i}$.

$$\mathcal{T}_e = \underset{i<\min(K,4)}{LCS}\left(\mathcal{T}_{e'_i}\right) + [P_e^n] \tag{III.1}$$

Since the LCS problem is NP-hard for an arbitrary number of traces, we use at most 4 traces, discarding the ones with a lower weight w_i. We can thus use

dynamic programming algorithm from Hirschberg (1975). A short history of other approaches to the LCS problem can be found in Bergroth et al. (2000).

Importantly, we compute the LCS on the sequence of predicate names, ignoring the role part. The roles in the output trace are set either to their original value if it was the same in all traces $\mathcal{T}_{e'_i}$, or the special *wildcard* value '_' otherwise. For instance:

$$
\begin{aligned}
&\mathit{LCS}\,(\\
&\qquad [\mathit{Circle}\,(1), \mathit{Extrude}\,(\text{FRONT}), \mathit{Extrude}\,(\_), \mathit{LoopCut}\,(1)]\,, \\
&\qquad [\mathit{Circle}\,(2), \mathit{Extrude}\,(\text{FRONT}), \mathit{LoopCut}\,(2)] \\
&) = [\mathit{Circle}\,(\_), \mathit{Extrude}\,(\text{FRONT}), \mathit{LoopCut}\,(\_)]
\end{aligned}
$$

NB The need need to relate the elements of the output mesh of a node to the elements of its input meshes is also a requirement of our DAG amendment of Section III.3.

III.2.5 Domain Specific Language for Selection Query

The language that we define for representing queries is composed of *patterns*, which select geometric elements based on their trace or spatial properties. Patterns are combined together using basic boolean operations, namely *Union* and *Difference*. In case no pattern-based query can be found, indices can be used to define a selection, thus falling back to the naive solution. The syntax of our query language is summarized as follow:

$$
\begin{aligned}
\mathit{query} :=\ & \mathit{MatchIndex}\,(\mathit{int}) \\
\mid\ & \mathit{MatchPattern}\,(\mathit{pattern}) \\
\mid\ & \mathit{Union}\,(\mathit{query}, \mathit{query}, \ldots) \\
\mid\ & \mathit{Difference}\,(\mathit{query}, \mathit{query})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{pattern} :=\ & \mathit{Predicate}\,(\mathit{role}, \mathit{role}, \ldots) \\
\mid\ & \mathit{Geometric}\,(\mathit{expression})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{role} :=\ & \mathit{int} \\
\mid\ & \mathit{symbol} \\
\mid\ & \mathit{wildcard}
\end{aligned}
$$

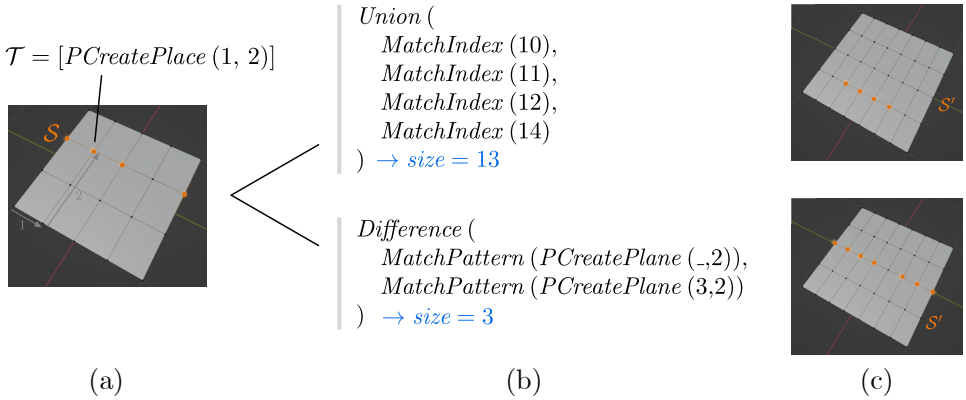


Figure III.2: *The example selection $\mathcal{S}$ from (a) can be matched by multiple queries Q like the examples of (b). These query generalize in different ways when the shape's hyper-parameters change, as shown in (c). We use a weighted size to define which one is the best option.*

Predicate-based patterns Such a pattern matches an element if the element's trace contains a predicate of the same name, with matching roles. For instance, the query *MatchPattern*(*Subdivide*) means to select all elements that were involved in a subdivide operation. This is refined when the predicate is parameterized with a role. Pattern's role parameters can be symbols or integers, or the special *wildcard* symbol to match any role.

Geometric patterns A geometric pattern selects elements based on spatial properties (proximity to a certain location, maximum in a given direction, etc.) or connectivity (number of corners to a face, number of neighbors, etc.). The syntax of the *expression* provided to a *Geometric* pattern varies whether the query is meant to select points, edges or faces. Our experiments do not make use of geometric pattern, we leave it here for future developments.

III.2.6 Query Synthesis

Synthesizing a query Q consists in **(i)** exploring the space of all programs that can be formed using the above-defined language, **(ii)** keeping the ones that respect the positive examples P and negative examples N and **(iii)** returning the program that is the most likely to be written by a human. The main challenges are that the program space is so huge that its exhaustive exploration is not possible (Section III.2.6.2), and the question of what a human would have programmed is ill-defined (Section III.2.6.1). We start with the later because knowing which programs are most relevant drives the way we optimize our program-space traversal.

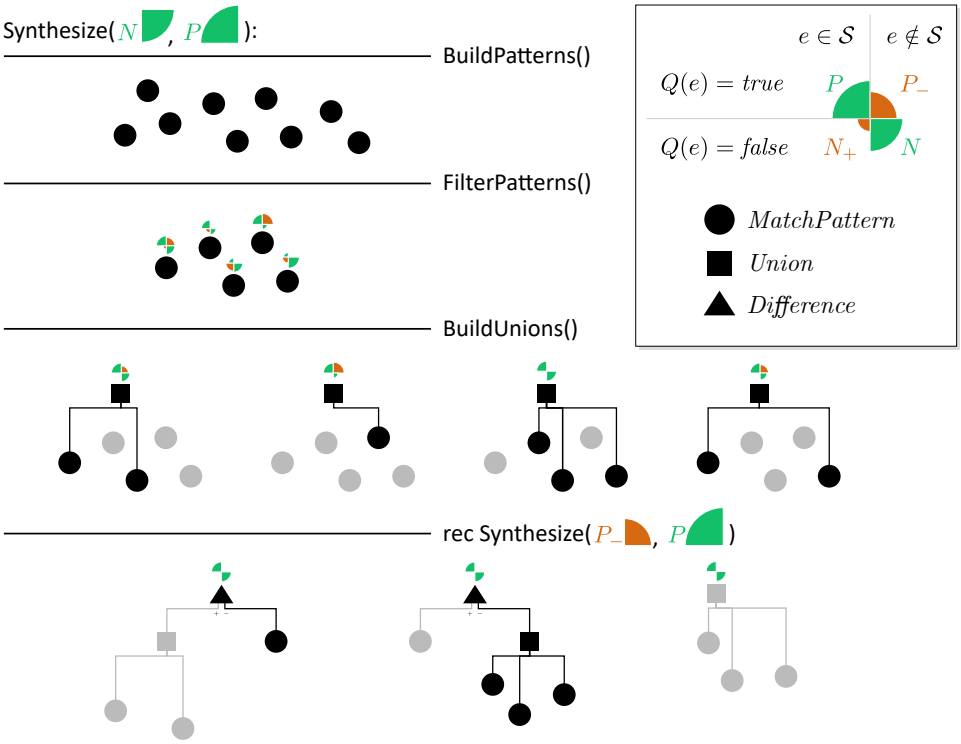


Figure III.3: *Main steps of our query synthesis method, presented in Algorithm 1.*

III.2.6.1 Best program selection

To distinguish which query the user intends to express among all queries that correctly match the input examples, we rely on a simple priors, known as *Ockham's razor*: the shorter the better. The shorter program that performs a task is the most likely to grasp the semantic intent that led to selecting elements. On the contrary, longer programs are in danger of over-fitting the examples, as illustrated in Figure III.2.

The size of a program is defined recursively as the number of nodes in its abstract syntax tree: a *MatchPattern* (...) has size 1, a *Union* (A, B) has size $1+size(A)+size(B)$, etc. We encode in this measure our intent to foster pattern-based selection over naive index-based selection by assigning a size of 3 to a *MatchIndex* (...) node.

III.2.6.2 Program space exploration

Given the grammar of our query language, we adopt bottom-up approach to generate queries. We first enumerate possible programs by starting from terminals nodes to build unions of patterns. Then we follow the spirit of synthesis through unification (Alur et al., 2015) to extend to the *Difference* operation by recursively synthesizing a query correcting false positives.

The main outline of our method is summarized in Algorithm 1 and illustrated in Figure III.3. It takes as input a set of positive examples $P = \mathcal{S}$ (traces that must be selected) and a set of negative examples N (traces that must *not* be selected). Throughout the algorithm, we also encounter *false positives* of a query q, which are elements that are matched by q but belong to N, and inversely *false negatives* of q, which are not matched by q but belong to P.

ALGORITHM 1: The outline of our recursive query synthesis algorithm. Calls (1) and (2) contain heuristics that reduce the program space exploration in order to speed up synthesis. Each union query returned by (2) comes with its set of false positives P_-.

```
Data: Positive examples P and negative examples N
Result: A query Q such that ∀p ∈ P, Q(p) = true and ∀n ∈ N, Q(n) = false
fn rec Synthesize P, N:
    patterns ← BuildPatterns(P, N);
    if FindPerfectPatterns(patterns) as p then
        return MatchPattern(p);
    end
    patterns ← FilterPatterns(patterns) ;                          (1)
    unions ← BuildPatternUnions() ;                                (2)
    candidates ← {};
    foreach (query, P_-) ∈ unions do
        if P_- ≠ ∅ then
            exception ← Synthesize(P_-, P);
            query ← Difference(query, exception);
        end
        candidates ← candidates + {query};
    end
    return argmin_{c∈candidates} size(c);
end
```

Pattern construction In the function `BuildPatterns()`, we build all predicate-based patterns that match at least one of the positive points. For each selected element $e \in P$, it iterates through all predicates $p \in \mathcal{T}_e$ and consider all patterns that match p. Given a predicate *Predicate*(a, b, c), matching patterns are generated by replacing any subset of the role parameters by the wildcard symbol e.g., *Predicate*(_, b, c), *Predicate*(_, b, _), etc. It returns a list of patterns together with their false positives P_- and false negatives N_+.

A pattern is called *perfect* if it has neither false positives nor false negatives. When one exists, it is an obvious solution to the program synthesis problem.

Pattern filtering When there is no perfect pattern, there are still ones that are better than others. A pattern that does not match all points but has no false positive is a good candidate to be part of a union-based query. A pattern that matches all

true positive points plus some false positives can be used in a difference-based query. More generally, we rank patterns based on the size of P_- and N_+:

$$score(pattern) = \begin{cases} +\infty & \text{if } P_- = \emptyset \\ +\infty & \text{if } N_+ = \emptyset \\ \dfrac{1}{|P_-| + \alpha|N_+|} & \text{otherwise} \end{cases} \tag{III.2}$$

The constant α represents the importance of false negatives relatively to false positive, we use $\alpha = 2$ in our experiments. The function `FilterPatterns()` eliminates all patterns whose score is bellow a threshold set to β time the highest non-infinite score. The value of β can be as low as 0 not to filter anything. This results in a broader exploration of the program space, which is slower but more likely to find the good query. A value higher than 1 means to only keep patterns that have either no false positives or no false negatives. In our experiments, we use $\beta = 1/3$.

Unions We combine the remaining patterns into *Union* queries that have no false negatives. A naive baseline would be to generate all subsets of the filtered collection of patterns, and keep all subsets whose intersection of N_+ is empty. However, there are too many such subsets in general, so here again we prioritize some combinations.

First, if it exists, a minimal union with no false positives can be built by greedily collecting the patterns that have the smallest N_+ among the ones that have no P_- . Such a union is a valid query for our synthesis problem.

Then we consider subsets of patterns by increasing cardinal. Subsets of cardinal 1 are patterns that have no false negatives. Subsets of size $k + 1$ are built from subsets of size k by adding a pattern only if it strictly reduces the number of false negatives. When this number falls to 0, the union is deemed *valid*. We rank valid unions using the same score as for pattern filtering and use the same threshold to eliminate the poorest unions. In order to limit the exponential number of recursive calls, we stop after a minimal amount of $\gamma = 8$ valid unions have been synthesized.

Recursion The last step of our synthesis process consists in building recursively, for each union u, a query e that matches all the false positives of u and none of its true positives. If e is found, $\mathit{Difference}(u, e)$ is a correct query for our initial problem. Elements not matched by u do not matter and are thus removed in the nested call, making it slightly faster.

We limit to recursion depth to $\delta = 3$; queries synthesized from a higher depth would in general feel convoluted to a human, stating selection rules based on exceptions of exception of exceptions etc.

III.2.7 Results

III.2.7.1 Experimentation

Setup We implemented our trace recording mechanism for vertices going through 6 types of DAG nodes:

- a grid generator, which marks the point at grid coordinate (i, j) with the predicate *PCreateGrid* (i, j);
- a circle generator, which orders points *PCreateCircle* (i);
- an extrusion, which marks points with either *PExtrude* (FRONT) when they are new points, *PExtrude* (BACK) when they are points from which new points have been added, or nothing for other points;
- a loop cut, which cuts face loops one or more times and marks points from the i-th cut with *PLoopCut* (i);
- a duplicate, which copies multiple times a part of the mesh and marks points from the i-th duplicate with *Duplicate* (i);
- a transform node, which move points according to a rigid transform matrix. Since it does not affect the connectivity of the mesh, we do not add any predicate.

Our semantic selection process is considered as a DAG node as well: the extrusion, the loop cut and transform operations use a selection parameter to restrict their effect to one area of the mesh only, and this selection can be driven by a semantic selection node.

Workflow The designer first builds a DAG. On one output of this DAG, they manually select some geometry, then validate to trigger our query synthesis routine. In a second time, they change the hyper-parameters of the graph and see whether the selection adapts correctly to the new shape. If not, they can manually fix the selection and refine the query. In such a case, the synthesis algorithm receives positive and negative examples from both instances on which there has been a manual selection and thus tries to build a query that works in both cases.

Synthesis Figure III.4 shows examples of results of our query synthesis method. The designer can audit the meaning of the selection query by looking at its human readable version (although this version of the query may be subject to ambiguities). The hyper-parameter that we vary is often the resolution of initial primitives since it is what most commonly break index-based selection.

Performance With our choice for parameters α, β and γ, we are able to run our query synthesis in a few hundreds of milliseconds. This enables us to synthesize a new query each time the designer changes the example selection. Figure III.6

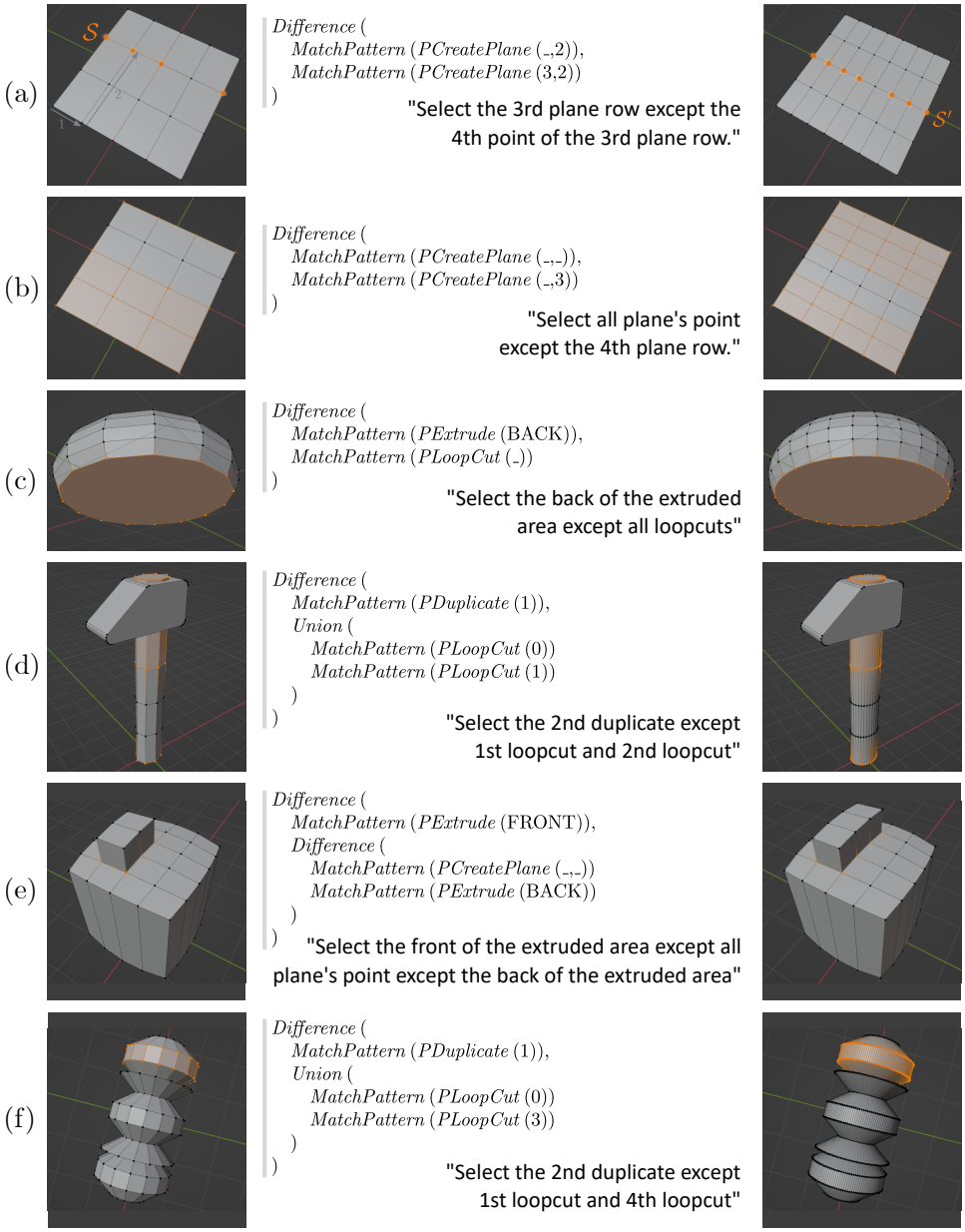


Figure III.4: *Examples of result of our query synthesis method. Left to right: the original user selection $\mathcal{S}$, the generated query Q, an automatically derived human-readable version of Q, and the selection $\mathcal{S}'$ after changing some hyper-parameter. The DAG corresponding to each shape is shown in Figure III.5.*

shows how this performance evolves when varying the parameters. Overall, the complexity is exponential in the recursive depth, but it is anyways unlikely that the query considered most appropriate by a human operator requires more than a

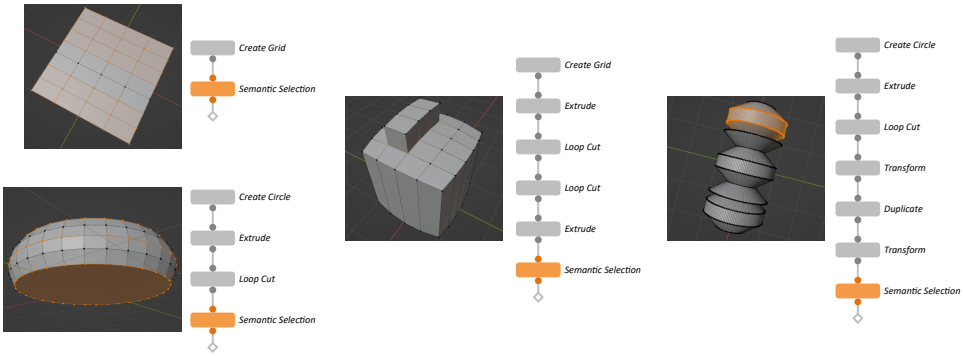


Figure III.5: *Generation DAGs used by the examples of Figure III.4. Selection inputs have been hidden to simplify display.*

few levels of recursion.

III.2.7.2 Discussion

Limitations Our method may fail in three different ways. The first one is illustrated in Figure III.7: the domain specific language we use to represent queries cannot express all the possible high level intents of the designer. This includes all selections based on geometrical features such as face orientation or proximity to a given location since we did not include the *Geometric*() predicate in our experiments.

Another case of failure is the one of queries that are too convoluted and thus optimized out by our pattern filtering method. A solution is to lower the α and β thresholds and increase the number δ of recursions, at the price of a longer synthesis (Figure III.6).

Lastly, our choice of returning the program of minimal size, among the ones that could be synthesized, may not match the intent of the designer. The possibility for the designer to correct the automatic selection on another example mitigates this issue though, and another solution could be to prompt them for the best human-readable query.

Analysis In a way, trace recording is a symbolic equivalent of automatic differentiation (Baydin et al., 2018), hinting about how to incrementally change the input to reach a given change in the output, so we believe that trace-based approaches and program synthesis are promising when applied to shape programs. Our query language model and synthesis strategy are fairly simple, but they already behaves in a much more meaningful way than the naive index based selection used in almost any tool.

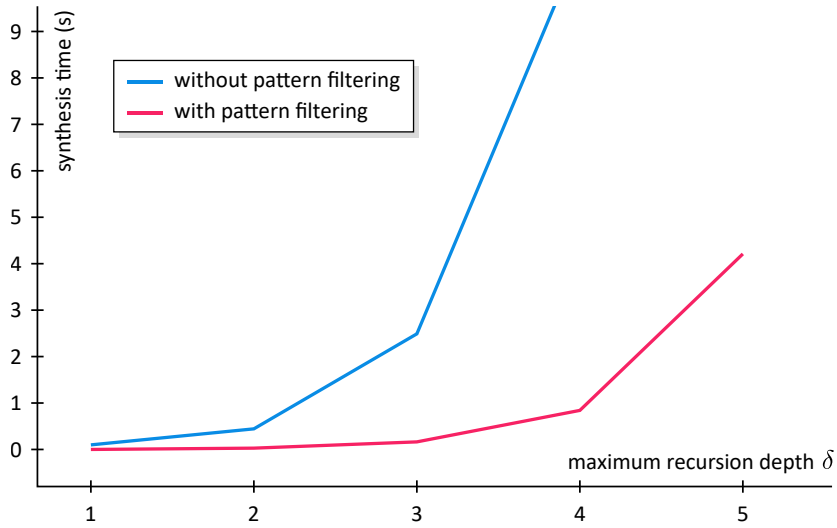


Figure III.6: *Synthesis time for the example of Figure III.4.f with varying recursion depths, with ($\beta = 1/3$) and without ($\beta = 0$) our pattern filtering method. The complexity remains exponential but becomes usable at interactive rates when filtering is enabled. Standard deviation is lower than the thickness of plot lines (on 10 samples).*

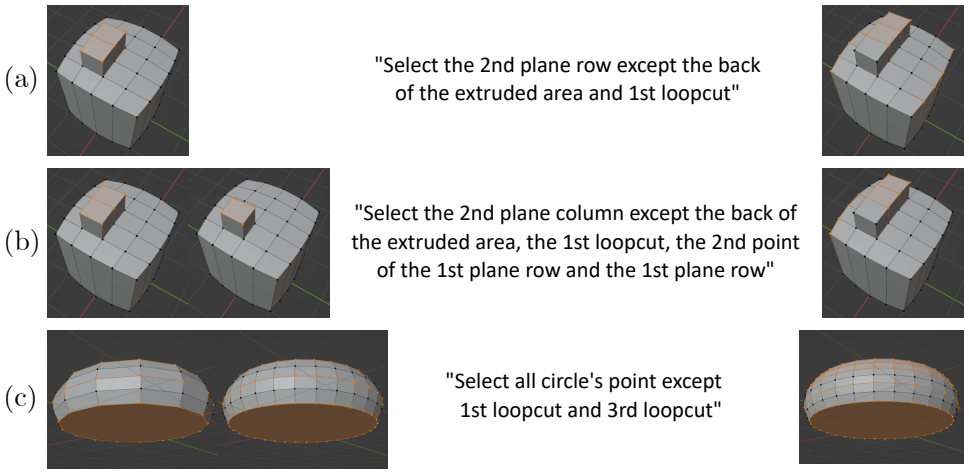


Figure III.7: *Cases of failure of our method. In (a) our system does not synthesize the expected query, which is to select only the front of the last extrusion, so we give another training example (b). It is still not enough because our DSL cannot distinguish the second extrusion from the first one. In example (c) the hyper-parameter is the number of loop cuts. Our DSL is not able to encode the notion of "middle" loop cut.*

III.2.8 Future Work

A wide range of possible improvement can be built upon our query synthesis approach. Some consist broadening the query language in order to encode a wider

diversity of intents, either by adding new constructs or introducing more powerful index expressions, others are related to improving the synthesis method. Both of these points would benefit for a closer comparison to the work of Mathur et al. (2020).

III.2.8.1 Variants of the query language

Spatial queries As mentioned already, the *Geometric* pattern would be an important improvement of the query language, since it is common for the designer to rely on spatial properties of the mesh elements, e.g., when selecting only faces that point upward. However this inflates the program space with many equivalently relevant queries to disambiguate from, and even make it continuously infinite when using inequalities on real numbers.

Neighborhood queries Our model of selection query is mostly declarative, like a SELECT query in SQL for instance. But we could combine it with imperative operations, in particular morphological operations (dilatation, erosion) which propagate the selected state to the direct neighbors of a mesh element.

Intersection We currently have no intersection operation in the query language. We could start by introducing first-level intersections only, to make synthesis easier. First-level intersections can only combine queries of the type *MatchPattern* or *MatchIndex*, but no *Difference* or *Union*.

A powerful construct to add is the *ordered intersection*. This is a first-level intersection which also checks that the patterns matched by its sub-queries appear in the very same order in the element's trace. This enables one to say "select points that were first part of a loop cut and *then* involved in an extrusion" and addresses the problem raised in Figure III.7.a and III.7.b.

III.2.8.2 Integer expressions

In the method we have described, integer roles are handled exactly as if they were abstract symbols. The numeric nature of these roles could be better used.

We could for instance allow queries to count downwards, thus querying for row $width - i$ in a grid of $width$ points wide. An easy integration of this feature is to extend the $PCreatePlane(i, j)$ to include $width - i$ and $height - j$ as extra roles.

The index may also be used to detect dashed selection. We can for instance use the following integer expression in patterns:

$$\begin{aligned} int := \ & constant \\ & \mid Modulo(int, int) \end{aligned}$$

The *constant* matches only one possible index, and *Modulo*(a, b) matches all indices i such that $(i - b) \equiv 0 \mod a$.

These integer expressions could be used in predicate matching, but also in *MatchIndex*, or even in a new construct *MatchIndexInSubquery*(int, query) that would combine semantic querying with a more arbitrary index-based picking at the end.

III.2.8.3 Variants of the synthesis algorithm

Programming by Demonstration We followed in this section the program synthesis principle of *Programming by Example*, but we could go one step further and do *Programming by Demonstration* by recording the different selection steps that the user follows (item by item, or loop select, etc.) instead of only using the resulting list of indices as example of output for our synthesis. This would allow the synthesis to be more constrained, thus faster and more likely to fit the user intent.

Outlier detection Our synthesis method assumes that the user-provided example is free of any error. We could however look for queries that are not perfect but close enough to the user input and thus hint the user for potential mistakes in their hand-selection.

Program space exploration Other techniques can replace our exponential recursion scheme, Mont-Carlo based exploration or decision trees. Quickly estimating an upper bound of the minimal query size would enable the use of adaptive values for thresholds α and β.

Evaluation Since our end result relies on the appreciation of the automatic selection by the user, we must conduct a user study to validate our approach and have an experimental evaluation of our performance. A key difficulty is to sample appropriately the space of shape programs that are actually used in practice.

III.3 Co-parameterization for the differentiation of parametric shape

III.3.1 Introduction

The philosophy of DAG amendment is to alter the node graph in order to augment the output geometry with extra information representative of the DAG execution. Two DAGs leading to the same geometry but using different processes may thus attach different extra information. Recording a full trace if a heavy process though, so we now explore a more lightweight DAG amendment, which only adds a pair of numbers to each face corner of the output mesh.

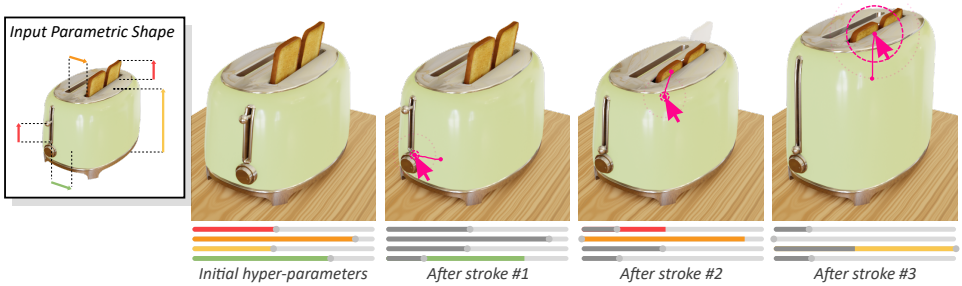


Figure III.8: *Our method infers without any manual setup how to update the hyper-parameters of a parametric shape to comply with an intent expressed as a brush stroke on its visualization. This enables a more direct and intuitive interaction process than tuning individual sliders, at no extra cost for the shape's designer.*

We still address what the CAD literature calls the naming problem, but this time at the scale of individual points of the surface of an instance. We assume a certain degree of regularity of the function $\mathcal{F} : \boldsymbol{\pi} \in \Pi \mapsto G \subset \mathbb{R}^3$, since it is intended for human interaction, so once we match points across multiple instances of a shape program we can thus measure finite differences.

Differentiation enables to locally inverse a function and has a wide range of application. In particular, in Section III.4 we use it to enable direct manipulation of a DAG's output geometry, back-propagating these changes to the input hyper-parameters.

Contributions Our key contribution in this section is an amendment operator for the parametric shape graph yielding a co-parametrization which associates points across hyper-parametric variations and thus makes it possible to measure point-wise shape jacobians efficiently.

Our approach is **(i) automated** – no extra effort is required from the shape's designer; **(ii) flexible** – it is possible to locally override the automated process whenever it is needed, and falling back to other methods remains possible at any time; **(iii) non-invasive** – it can fit into existing parametric shape engines without requiring to rewrite the content of generation operations.

Hypotheses Although we tried to remain agnostic in the underlying DAG engine – in particular we do not require it to be automatically differentiable – we make the assumptions that the operations (a) process only mesh-based data (b) can transmit extra attributes of the kind of texture coordinate from their input to their output and (c) label the output geometry with a duplicate index (that we denote j) when they duplicate input geometry.

III.3.1.1 Problem Setting

Let P be a point of an instance $G = \mathcal{F}(\boldsymbol{\pi})$ of the parametric shape $\mathcal{F}$. We would like to measure the *influence* of the hyper-parameters $\boldsymbol{\pi}$ on P, as this is in essence what differentiating means. Following the usual finite-difference based definition of the differential, we consider $G' = \mathcal{F}(\boldsymbol{\pi}')$, where $\boldsymbol{\pi}' = \boldsymbol{\pi} + d\boldsymbol{\pi}$ is infinitesimally close to the original hyper-parameter $\boldsymbol{\pi}$. In other terms, this consists in evaluating the position of the point for two close enough values of an hyper-parameter and measuring their difference. If we note P' the equivalent of P in G', the definition of the derivative of $\mathcal{F}$ in a direction v could be, assuming that $d\boldsymbol{\pi} \propto v$:

$$D_v(\mathcal{F})(\boldsymbol{\pi}) \simeq \lim_{\|d\boldsymbol{\pi}\| \to 0} \frac{P' - P}{\|d\boldsymbol{\pi}\|} \tag{III.3}$$

Even if $\mathcal{F}$ is regular enough for the limit to converge, there is a critical omission in this definition: how do we define P'? The *equivalent* of P in G' is an ill-defined notion.

As long as the point P is only identified by its 3D location, the best we can do is to look at its relation to other points in the same shape. This is at the heart of the field of *shape correspondence* techniques. But this approach requires heavy geometry analysis processes which are incompatible with interactive applications, and misses the semantic information contained in the structure of the DAG.

Our approach consists in augmenting the output of $\mathcal{F}$ so that each point comes equipped with an identification information that can be used to quickly "recognize" P in G'. This information is called the *co-parameter* a_P of P and P'. As illustrated on a simplified example in Figure III.9, there are multiple ways to co-parameterize a parametric shape. In our work we leverage the information contained in the DAG to pick one that is relevant to the designer's use case.

Terminology Here are the key terms we use along this section.

A *Parametric shape* $\mathcal{F}$ is a function mapping input values $\boldsymbol{\pi}$ called *hyper-parameters* to 3D surface meshes. An *instance* of the parametric shape is this 3D surface mesh for a fixed value of the hyper-parameters.

A *Single-point parametric subshape* takes as input the same hyper-parameters than the original parametric shape, but only outputs a single point from the corresponding instance. It may be undefined for some values of the hyper-parameters, otherwise returns point that has the same *meaning*.

The *parameter* of a 3D point of an instance designates a 2D coordinate that indexes this point and is often used for texture mapping.

The *co-parameter* a of a single-point parametric subshape is a coordinate that indexes this subshapes among all the other ones. By extension, the co-parameter

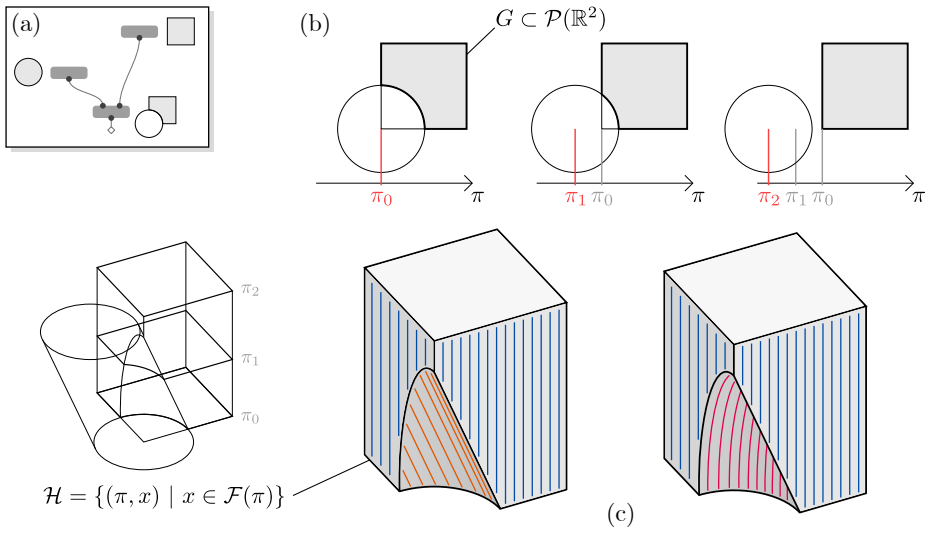


Figure III.9: *(a) A simple 2D parametric shape with a single hyper-parameter π and (b) some of it's instances. A co-parameterization of $\mathcal{F}$ is a parameterization of its graph $\mathcal{H}$ in dimensions $dim(\Pi) + dim(\mathbb{R}^2)$, and as shown in (c) multiple co-parameterization of a same shape are possible.*

of a point of an instance is the co-parameter of the single-point parametric subshape that this point is an instance of.

III.3.1.2 Related Work

Shape correspondence We mentioned our need to identify points across multiple meshes of potentially varying connectivity; this is commonly referred to as *shape correspondence* or *cross-parameterization* (Schreiner et al., 2004; Kraevoy & Sheffer, 2004; Kilian et al., 2007). It consists in mapping each point from a shape to points that have the same *semantic* but in other shapes.

van Kaick et al. (2011) surveys a large variety of shape correspondence works, and more recent work even try to match dissimilar shapes (Hecher et al., 2018). But this field focuses generally on offline registration of a small number of geometries, while we have to register a continuous infinity of meshes. Some works build correspondences for large amount of objects. Mahmood et al. (2019) addresses the lack of consistent parameterization among datasets of human bodies, but is hand tuned for this very use case. Leimer et al. (2017) creates a parametric shape by registering together a whole collection of shapes. Unfortunately these methods are offline and resource intensive. Furthermore there are no geometric features guaranteed that we may rely on to in general. For all these reasons, we adopt a quite different approach. Establishing a shape correspondence is a semantic operation, so we leverage the implementation of the parametric shape – the DAG – because its structure carries semantic information beyond what the output geometry shows.

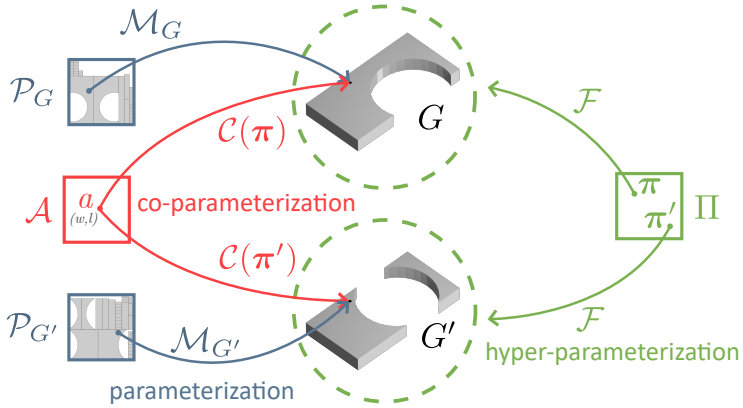


Figure III.10: *We need to* recognize *a point after a change of the hyper-parameters $\boldsymbol{\pi}$. We model this using three notions of parameterization. $\mathcal{M}_G : \mathcal{P}_G \to G \subset \mathbb{R}^3$ is a parameterization as meant in parameterized surfaces. The parametric shape $\mathcal{F} : \Pi \to \{G\}$ itself is a higher-order parameterization. Since $\mathcal{M}_G$ is not enough because in general it is different for each $\boldsymbol{\pi}$, we introduce $\mathcal{C} : \Pi \to (A \to \mathbb{R}^3)$ which outputs parameterizations consistent among all the geometries resulting from $\mathcal{F}$.*

III.3.2 Co-parameterization

III.3.2.1 Co-parameter definition

As highlighted in Section III.3.1.1, the jacobian of a parametric shape $\mathcal{F}$ is ill-defined, because the function $\mathcal{F} : \boldsymbol{\pi} \mapsto G$ returns a set of many points – an infinity of points – with no way to *recognize* one among them. Any differentiation scheme, be it automatic, suffers from this definition issue. We thus introduce the notion of *co-parameterization* of $\mathcal{F}$, a way to extract what we call *single-point parametric subshapes* of the form $\boldsymbol{\pi} \mapsto x \in \mathbb{R}^3$. Contrary to $\mathcal{F}$, these subshapes can be differentiated.

The usual way to identify a point on a geometry is to *parameterize* it. Importantly, this must not be confused with our hyper-parameterization (see Figure III.10). It consists in defining for a fixed geometry G a bijection $\mathcal{M}_G : \mathcal{P}_G \to G$ mapping to each point of G a *parameter* from a set $\mathcal{P}_G$. Such a parameter can typically be a unique texture coordinate or – in the case of meshes – a face index together with barycentric coordinates. There are in general many different ways of parameterizing a given geometry.

The problem in our case is that this mapping $\mathcal{M}_G$ may depend on $G = \mathcal{F}(\boldsymbol{\pi})$, and so on the hyper-parameter $\boldsymbol{\pi}$. As a consequence, it is of no use to *recognize* a point after $\boldsymbol{\pi}$ changed. Hence the need for a collection $\mathcal{C}$ of *consistent* parameterizations, each associated with a different $\boldsymbol{\pi}$ but all sharing the same parameter set $\mathcal{A}$:

$$\mathcal{C}(\boldsymbol{\pi}) : \mathcal{A} \longrightarrow \mathcal{F}(\boldsymbol{\pi}) \subset \mathbb{R}^3$$

The strength of this second order function C is that it may be uncurried because $\mathcal{A}$ does not depend in $\boldsymbol{\pi}$, so

$$C : \Pi \longrightarrow (\mathcal{A} \longrightarrow \mathbb{R}^3)$$

becomes

$$\tilde{C} : \Pi \times \mathcal{A} \longrightarrow \mathbb{R}^3$$

and may even be curried back with its arguments swapped:

$$\bar{C} : \mathcal{A} \longrightarrow (\Pi \longrightarrow \mathbb{R}^3)$$

We call $\bar{C}$ a *co-parameterization* of the parametric shape $\mathcal{F}$ and $\mathcal{A}$ its *co-parameter set*. It plays a role similar to the surface parameterization but in the space of parametric shapes. With these notations, for each $a \in \mathcal{A}$ the function $\bar{C}(a)$ is a differentiable object, for which using for instance finite differences makes sense. We call this a *single-point parametric subshape* of $\mathcal{F}$ (the output of step 1 in Figure III.21).

So, to determine the influence of the hyper-parameters on a point p_i sampled on the geometry $G = \mathcal{F}(\boldsymbol{\pi})$, we actually consider its co-parameter $a_i = C(\boldsymbol{\pi})^{-1}(p_i)$ and evaluate the jacobian $J_i(\boldsymbol{\pi})$ of $\bar{C}(a_i)$ at $\boldsymbol{\pi}$. The co-parameter a_i of a point p_i is thus the way to "recognize" it after a change of the hyper-parameters. We discuss in the next section how to build this co-parameterization in practice.

III.3.2.2 Automatic DAG Amendment

We assume in this section that the geometry produced by the parametric shape $\mathcal{F}$ is a 3D surface mesh. We automatically modify $\mathcal{F}$ so that the geometries it produces have each of their face corners labeled with their co-parameter (Figure III.12). Thus, sampling a 3D point p_i onto the output mesh also provides its co-parameter $a_i = C(\boldsymbol{\pi})^{-1}(p_i)$ by interpolating the co-parameters of the corners of the face that p_i belongs to (Figure III.22.a).

Without loss of generality, we can model the implementation of $\mathcal{F}$ as a DAG whose nodes are mesh processing operations. Hyper-parameters affect the behavior of individual nodes, but the connectivity of this graph remains static. Our automatic modification of $\mathcal{F}$ consists in inserting new nodes into this graph. It is non-invasive in the sense that it does not require to bring any change to the internal logic of individual nodes. The co-parameter attribute a that we intend to create at each face corner must be:

- **unambiguous** – There must not be two points sharing the same value of a.

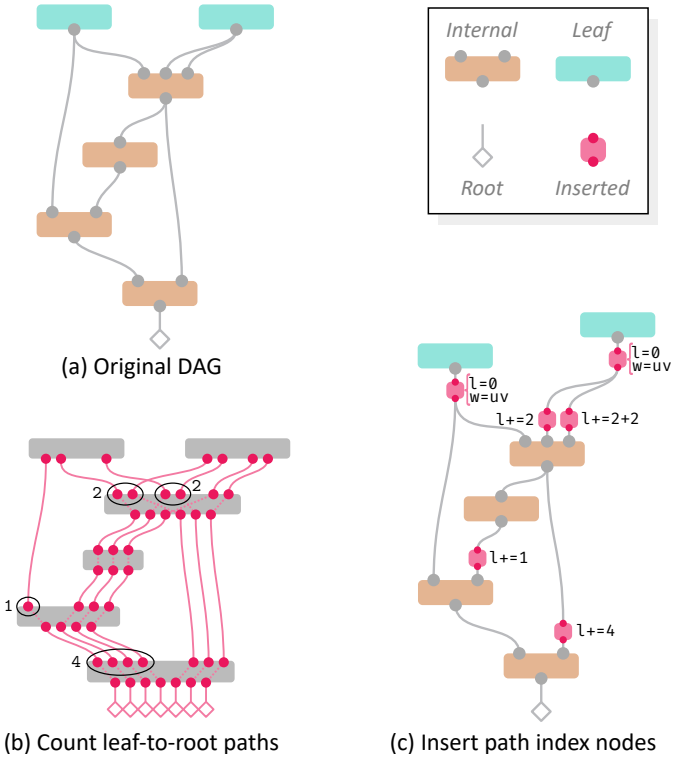


Figure III.11: *We identify points across different outputs of a DAG (modeling the implementation of the parametric shape) using two attributes attached to face corners. w is a copy of the parameterization (UV) at the leaf the face corner comes from. l is a unique index of the leaf-to-root path that generated the face. We first count the number of paths flowing through each input of each node (b). We then automatically insert nodes (c) to first initialize l to 0 after each leaf and offset l before any input by the number of paths flowing through previous inputs of the same node. As a result, each face corner of the output geometry is labeled with a unique path index.*

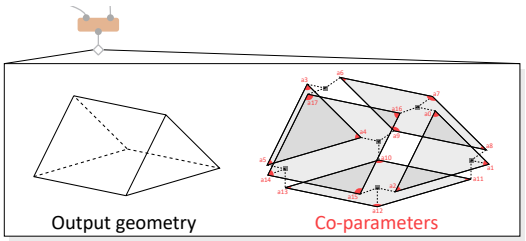


Figure III.12: *The output of the DAG is augmented with a co-parameter labeling each face corner.*

- **interpolable** within a face – In order to infer any point's co-parameter from the value at the corners of its face.

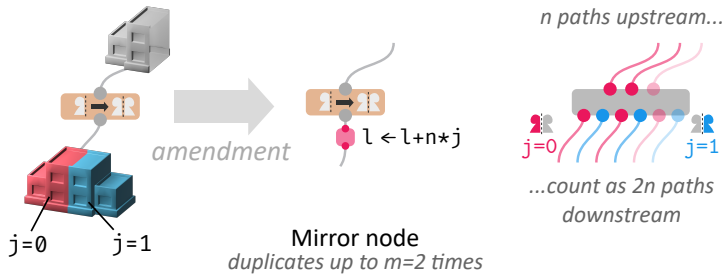


Figure III.13: *A node that duplicates geometry up to m times and has n incoming paths is considered downstream as being traversed by $n \cdot m$ paths. Assuming there is a way to infer the index j of the duplicate a face belongs to, the path index l is replaced by $n \cdot j + l$.*

- **consistent** across possible values of the hyper-parameters – To ensure the continuity of the single-point parametric subshapes $\boldsymbol{\pi} \mapsto \mathbb{R}^3$ that we extract.

We split a into a real component w and an integer component l. The real component is technically no different from a texture coordinate, which is also a real vector attached to face corners. The integer component must be constant across a given face in order to ensure interpolability, so it may in practice be attached to faces rather than corners.

The attribute l of a face contains the index of the data flow path that generated it (Figure III.11.b). This information is consistent since the connectivity of the DAG never changes once the shape has been modeled. Disambiguating faces generated through the same path is ensured by the real component w that is given by a standard parameterization of the leaf of this path.

Construction We first insert a node after each leaf of the DAG. This node initializes w by copying the canonical texture coordinate output by the leaf. When the leaf node generates meshes of constant connectivity, any fixed automatic parameterization (auto UV unwrapping) can be used. When the node is a primitive shape (sphere, cylinder, box, etc.), its canonical parameterization works. Practical mesh-based parametric shape engines support forwarding face corner attributes through their internal nodes like any other texture coordinate, so w is hence defined at the output of the DAG.

To produce the integer part, we first initialize it to $l = 0$ after each leaf (in the same node that initializes w). Then, before the k-th input of an internal node, we add a node that increments l by $\sum_{i<k} n_i$ where n_i is the number of paths going through input i (see Figure III.11).

The goal of the index l is to disambiguate cases where w overlaps. Counting paths is a way to address cases caused by nodes that combine several input meshes, like a boolean operation (difference, fusion, intersection). The other major source

ALGORITHM 2: Our DAG rewriting algorithm.

```
CountPaths(dag.root);
for n ∈ dag.nodes do
    c ← GetMaxDuplicates(n);
    if IsLeaf(n) then
        InsertAfter(n, MakeInitNode());
    else if c > 1 then
        InsertAfter(n, MakePostDuplicateNode(c));
    sum ← 0;
    for input ∈ n.inputs do
        if input.index > 0 then
            InsertAfter(n, MakeIncrementNode(sum));
        end
        sum ← sum + input.path_count;
    end
end
```

of overlap is duplication nodes (mirror, copy and transform, scatter, etc.). To include this into our framework, we multiply the number n of paths flowing into a duplication node by the maximum number m of duplicates it may produce (Figure III.13). If the duplication index j has a finite number of m possible values (for a mirror, $j \in \{0, 1\}$), we insert after the duplication a node that replaces l with $n \cdot j + l$.

If the duplication index j may take an arbitrary large value, we add an extra dimension to the integer index l to store it, promoting it to an integer vector. Since the real component w is typically in $[0, 1]^2$ the first two dimensions of l are emulated by offsetting w in order to alleviate memory usage.

Thus, each face corner of the output of the DAG is uniquely and consistently identified by its path index l and leaf parameter w. Our process is summarized by pseudo-code in Algorithms 2 and 3.

III.3.3 Results

III.3.3.1 Implementation

Our DAG automatic amendment (Section III.3.2.2) is exemplified in Figures III.14 and III.15, show the original DAG and its amendment for example shown later in Section III.4.3. Additional results are available in Appendix A.

Performances are reported together with the interactive results later on in Section III.4.3. The relevance of amendments and thus of the jacobians we compute is also evaluated in the interactive section by pooling users about their feeling when using the resulting tool.

ALGORITHM 3: The recursive pseudo code of `CountPaths`. We memoize the result at each node input in the field `path_count`.

Input: Some DAG node n
Output: The number count of path flowing through this node
if `IsLeaf(`n`)` **then**
    count ← 1;
else
    count ← 0;
    for input ∈ *n.inputs* **do**
        if input.*path_count is not defined* **then**
            input.*path_count* ← `CountPaths(`input.*connected_node*`)`;
        end
        count ← count + input.*path_count*;
    end
    count ← count · `GetMaxDuplicates(`n`)`;
end

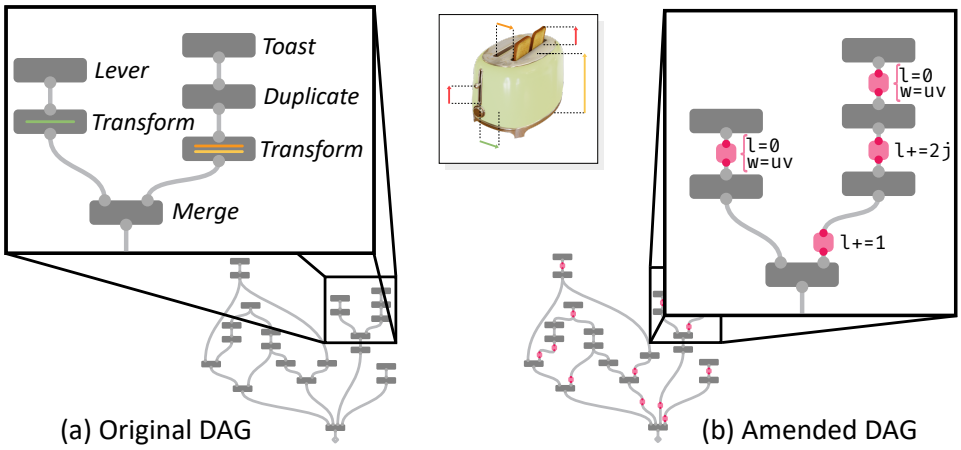


Figure III.14: *Preview of the DAG amendment applied on the example of Figure III.8. Small pink nodes in (b) are the node we insert. A more detailed version of this figure can be found in the supplementary material.*

NB Our process does not conflict with texture mapping (Figure III.16). The real component w of the co-parameter is in nature similar to a texture coordinate but in an extra layer. The texture mapping of the table example received extra care from the designer so that the wood look of the plate is extended rather than stretched when the size-related hyper-parameters are edited.

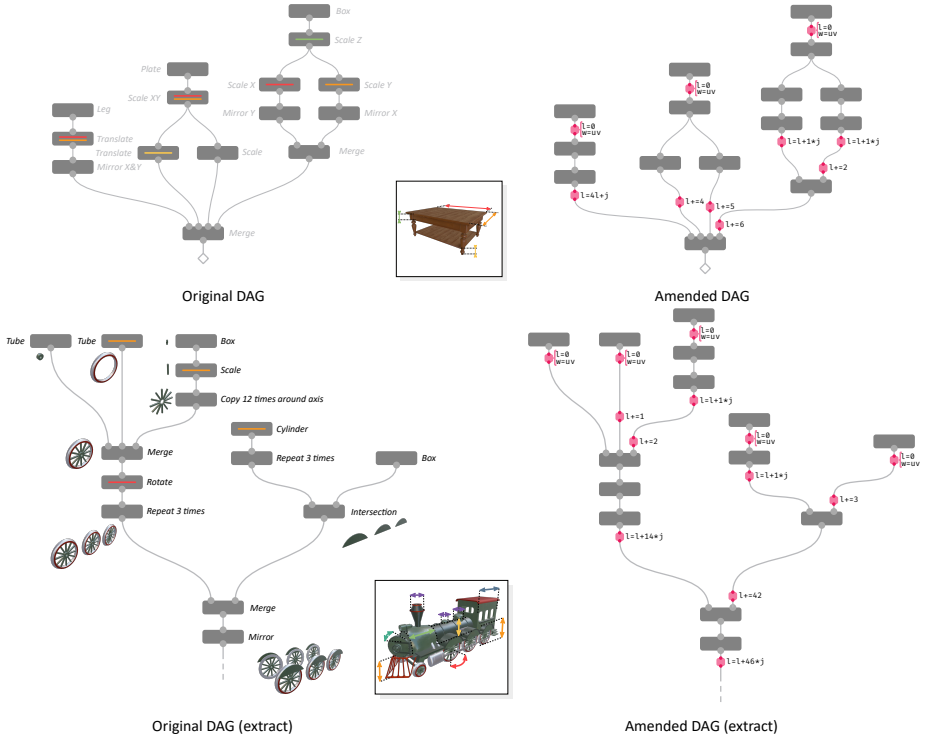


Figure III.15: *Original DAG (left) and our automatic DAG Amendment (right) for the example (c) and (e) of Figure III.24. Colored lines show the hyper-parameters influencing an individual operation.*

Figure III.16: *Although the real component w of the co-parameter behaves like a texture coordinate, it is stored in an extra attribute so it does not prevent the shape's designer to customize texture mapping.*

III.3.3.2 Limitations

Co-parameterization Our proposed model of co-parameterization relies on the practical ability of the DAG nodes to forward extra attributes on face corners. Although this can be seen as a restriction, it is a very reasonable assumption

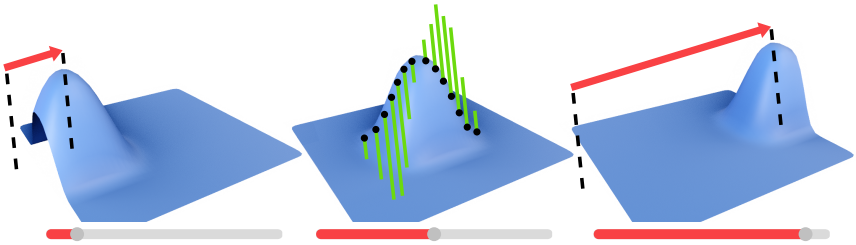


Figure III.17: ***Limitation:*** *The only operation of this parametric shape consists in moving a vertex and its neighborhood. The hyper-parameter defines which vertex is selected rather than how to move it, so the Jacobians of single-point subshapes (lines shown in the middle figure) do not match the intuitive influence of the hyper-parameter.*

provided that production-ready parametric shape engines usually need this feature in order to conserve texture coordinates.

Some nodes though introduce overlaps in UVs even when there was none in their input (e.g; some smoothing algorithms). Some other nodes are simply not able to assign face corner attributes to their output (e.g; a convex hull node). Discrete hyper-parameters like the number of repetitions, in a duplication operation, are not handled by our approach as is because it makes the single-point subshapes non differentiable.

It is nonetheless always possible for the shape's designer to manually overcome these cases by adding extra nodes dedicated to fixing the w attribute. Figure III.18 shows an example where a continuous box proxy is used to override the value of w after a duplication.

Unintuitive jacobians When an hyper-parameter acts on the selection from a geometry that gets affected by an operation rather than the way the operation itself moves points, the jacobians of single-point subshapes may no longer match the intuition of the end-user (see Figure III.17).

III.3.3.3 Future Work

More semantic One of the strengths of our approach is to leverage the semantic information carried by the DAG. One could look for other ways of using it. For instance, the depths of the nodes using a particular hyper-parameter could be used to prioritize some of them during jacobian filtering (Section III.4).

Proxies In cases where limitations occur in the construction of the co-parameterization, hand-tuned workarounds based on geometry proxies are possible. We could explore ways to automate this. For instance, in Figure III.18 a continuous box proxy is used to override the value of w after a node on discontinuous operations. The

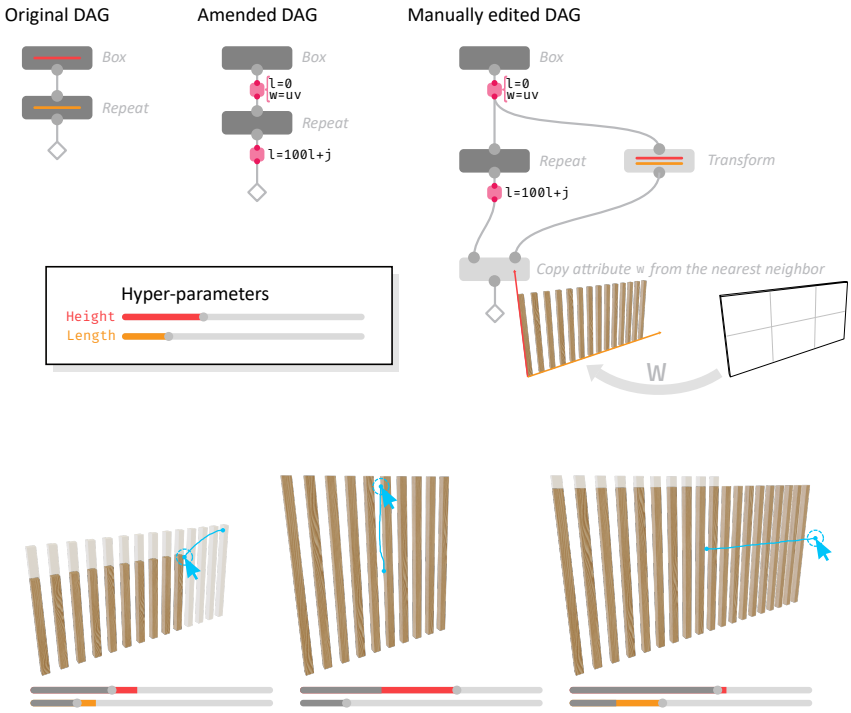


Figure III.18: *When one of the hyper-parameter has a non continuous influence like the length here – acting on the number of repetitions – the designer may manually amend the DAG after our process to make the real component w of the co-parameter continuous anyway, for instance by projecting it from a box proxy.*

Repeat node of this examples floors the `length` hyper-parameter, which prevents the output from being differentiated after our regular DAG amendment. The proxy is used to provide back continuity to the single point subshapes that are sampled at the beginning of an interaction.

DAG pruning The restriction $\bar{C}(a)$ of the parametric shape $\mathcal{F}$ to the single point of co-parameter $a \in \mathcal{A}$ may not be affected by all the nodes of the DAG. The graph could hence be pruned while measuring finite differences in order to alleviate its evaluation cost.

Auto-differentiation Using automatic differentiation can make the nodes of the DAG output a jacobian as part of their computation process. This replaces the time consuming evaluation of finite differences and also enables to update the jacobian buffer at each frame during a stroke. Cascaval et al. (2022) use this to better scale to shapes with tenths of hyper-parameters, but their framework is limited to constant connectivity, as advanced mesh processing nodes like boolean operations are non trivial to implement using an automatic differentiation framework.

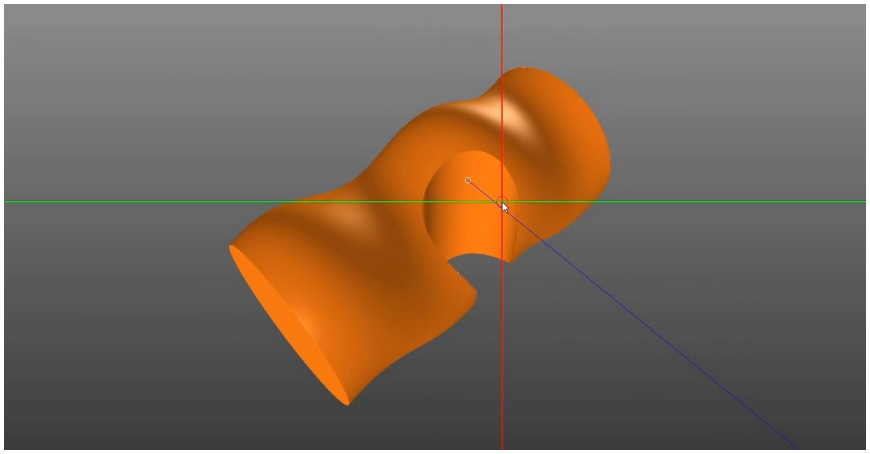


Figure III.19: *A prototype of our method applied to a shape represented as a signed distance field.*

Adaptation to implicit representations of surfaces Our practical construction of the co-parameter $a = (w, i)$ focuses on mesh-based representation of 3D surfaces, but the overall approach and the notion of co-parameterization is more general. We show for instance in Figure III.19 an implicit surface rendered using sphere tracing for which we amended the signed distance function to also return a path index l and leaf parameter w. The evaluation of the position of a point given its co-parameter is done by a secondary program, derived from the signed distance function (manually, in this prototype).

III.4 Jacobian Filtering: Applying Inverse Kinematics to Parametric Shapes

The core problem of parametric shape manipulation by the end user is that it does not occurs in the 3D space but rather in the hyper-parameter space Π, which is roughly a list of sliders in a user interface. Yet, the *intent* of the end-user is often more naturally expressed in the 3D space. This mismatch results in a trial and error loop that is dampening the creation process.

In some contexts like animation, this issue is such a deal-breaker than riggers are asked to equip shapes with *manipulators*, which are handles lying in the 3D space and whose transform drives some hyper-parameters. But creating these requires extra time and skills on top of the design of the parametric shape itself.

In the workflow we target (Figure III.20), a technical designer first builds an object using non-destructive modeling tools, and simply exposes some hyper-parameters without minding manipulators, thus defining $\mathcal{F}$. Lastly, the end-user edits the hyper-parameters to customize the object. In between, our DAG amendment

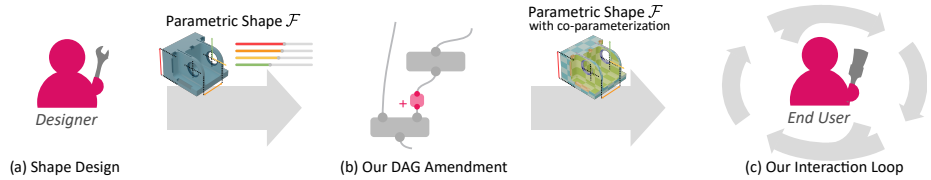


Figure III.20: *A common creation workflow separates (a) the designer of a parametric object from (c) its end user. The former handles technical issues for the latter to be fully dedicated to more artistic and intuition based matters (e.g; animation, staging). Our method improves this end interaction without extra effort from the designer by (b) automatically inserting a few nodes to the parametric shape's graph representation (DAG) produced by the designer.*

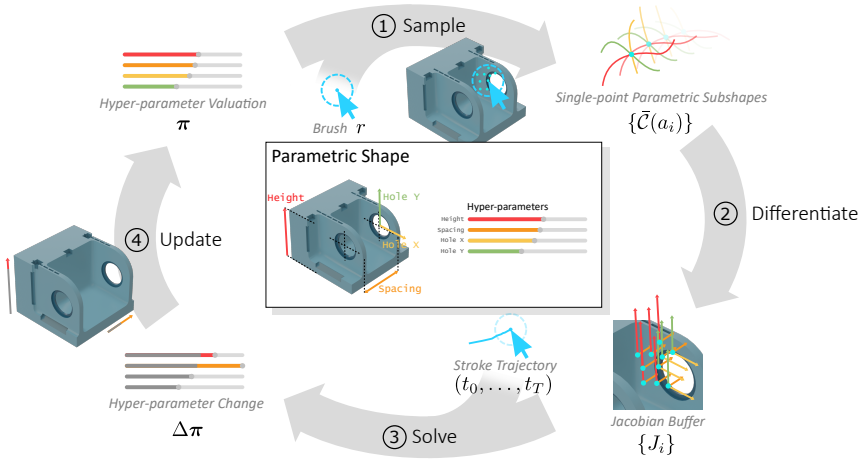


Figure III.21: *Overview of our interaction loop. At the beginning of a stroke, points are sampled around the user cursor to extract co-parameters a_i and so single-point parametric shapes $\bar{\mathcal{C}}(a_i)$. Each of these is differentiated to measure their jacobians, which are then provided to the solver. Confronting jacobians to the trajectory of the cursor, the solver determines the update to apply to the hyper-parameters.*

automatically modifies $\mathcal{F}$ so that besides the hyper-parameters sliders the end user may directly manipulate the shape in the 3D view.

In this section, we introduce a method for interpreting user inputs expressed in the 3D viewport as changes in the hyper-parameter space without any extra controller setup. Our key contribution is a non-linear filtering mechanism acting on the the resulting jacobians to both regularize and sparsify the shape optimization, fostering hyper-parameters whose behavior comply with the scale of the user brush.

III.4.1 Introduction

III.4.1.1 Overview

Following a common painting metaphor, we model the user input as a series of brush strokes. Each of these strokes must be interpreted as a modification $\Delta\boldsymbol{\pi}$ of the hyper-parameters. This grounds the interaction loop shown in Figure III.20.c and detailed in Figure III.21. Our loop follows the usual approach of inverse control problems, namely getting a differential information (Jacobian matrices $\{J_i\}$) in order to locally inverse the function $\mathcal{F}$ (the solver). For the solving part we can draw from the IK literature, however this literature takes for granted the access to the Jacobians, which is not obvious in our case.

In the previous section, we focused on how to theoretically define and practically measure the Jacobians telling the influence of the hyper-parameters over the part of the geometry where the stroke starts. Section III.4.2 shows how we use this differential information to compute $\Delta\boldsymbol{\pi}$ and details the choices we have made compared to other such solving contexts. We then show results in Section III.4.3 and finally discuss the current limitations and many prospects of our method in Section III.4.4.

III.4.1.2 Sampling and differentiation

At this point we are able to define what the jacobian of a point p_i sampled on the geometry $G = \mathcal{F}(\boldsymbol{\pi})$ means. When a stroke starts, we sample K such points by casting rays from the viewport and intersecting them with G. The hit information is used to not only find the 3D intersection point p_i but also its co-parameter $a_i = (w_i, l_i)$.

The extent of the brush may cover areas at very different depths, but we assume that the user intent has a limited depth of field, affecting either the foreground or the background but not both at the same time. To match this, we select the closest sample to the center of the brush, and discard all the points that are too far from its unprojected world space location.

To measure the k-th column of the jacobians J_i, we evaluate the parametric shape with the k-th hyper-parameter slightly changed. The step of differentiation is set to $\delta_k = 10^{-5} \cdot (\alpha_k - \beta_k)$, where $[\alpha_k, \beta_k]$ is the range of possible values of this hyper-parameter. Within the new geometry G' that this produces, we look for points that have their co-parameter equal to a_i. For each possible value of l_i, we build a mesh where coordinates are the w attribute of face corners from G'. We then project w_i onto this mesh to find the face index and barycentric coordinates of the new position p'_i of the i-th sample with respect to G'. The k-th column of J_i is thus $(p'_i - p_i)/\delta_k$ (see Figure III.22).

If the nearest neighbor of w_i is too different, we assume that the point p_i has no equivalent in the new geometry G'. This happens for instance for points at the

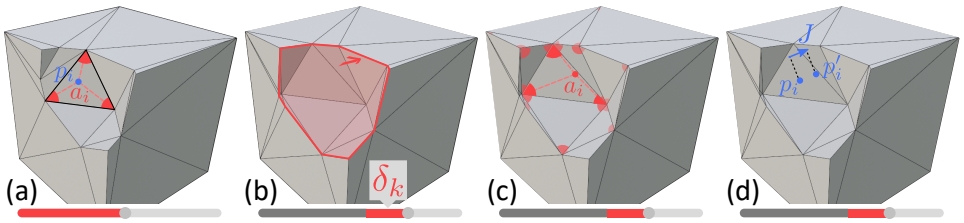


Figure III.22: *To evaluate a column of the jacobian at a sample point p_i, (a) we use its co-parameter a_i interpolated from the face corners, then (b) vary the hyper-parameter by δ_k and (c) look for the new point p'_i whose co-parameter equals a_i. (d) The column of the jacobian w.r.t. this hyper-parameter is $(p'_i - p_i)/\delta_k$.*

edge between the operands of a boolean operation. In such a case, we set the k-th column of J_i to zero to prevent changing this hyper-parameter, provided we do not know its influence.

III.4.2 Solving

The solver is provided with the jacobians $J_i \in \mathbb{R}^{3\times n}$ measured at the K points p_i sampled within the brush of radius r when the stroke started as well as the trajectory $(t_0, \ldots, t_T)$ of the stroke. The solution $\Delta\boldsymbol{\pi}$ returned by the solver must ensure the following properties:

exactness The points originally lying inside the brush must still be inside the brush at the end of the stroke.

sparsity The hyper-parameter update must have an amplitude as low as possible; the user does not expect a single stroke to apply too significant changes.

continuity The hyper-parameter update must be continuous along the trajectory, i.e., adding a new way point t_{T+1} close to t_T must not suddenly change $\Delta\boldsymbol{\pi}$.

speed A result must be found at interactive frame rate. The user should not feel that hyper-parameters are changing while they are not moving the mouse.

III.4.2.1 Inversion

At the first order, we know that for each of the single-point parametric subshapes $\bar{C}(a_i)$ that we sample – denoted simply $\bar{C}_i$ below – we can approximate the new location of the point using the jacobian $J_i = J_{\bar{C}_i}(\boldsymbol{\pi})$ computed at step 2 of Figure III.21:

$$\bar{C}_i(\boldsymbol{\pi} + \Delta\boldsymbol{\pi}) \simeq \bar{C}_i(\boldsymbol{\pi}) + J_i \cdot \Delta\boldsymbol{\pi} \tag{III.4}$$

The stroke trajectory is expressed in the viewport, so we compose equation III.4

with a function $Proj : \mathbb{R}^3 \to \mathbb{R}^2$ mapping the world space to the screen space. Since $\bar{C}_i(\boldsymbol{\pi}) = p_i$ is the point that was clicked on, it is mapped to t_0 – the beginning of the stroke. To fulfill the objective of exactness, we want the new position $\bar{C}_i(\boldsymbol{\pi} + \Delta\boldsymbol{\pi})$ of this point to match the new position t_T of the user's cursor:

$$t_T = t_0 + J_i' \cdot \Delta\boldsymbol{\pi}$$

where $J_i' = J_{Proj} \cdot J_i$ is the jacobian of the composition with the projection.

This is a typical problem of inverse kinematics which can be solved with a damped least square method (Deo & Walker, 1992; Baerlocher & Boulic, 2004). Such a method finds the solution $\Delta\boldsymbol{\pi}$ that has a near minimal L_2 norm while avoiding discontinuities at singularities (where the rank of J_i' changes):

$$\Delta\boldsymbol{\pi} = J_i'^{+} \cdot \Delta t$$

where $\Delta t = t_T - t_0$ and $J_i'^{+}$ is a singularity robust pseudo-inverse of J_i'.

> **NB** *Jacobian J_{Proj} of the projector.* Let $Proj : \mathbb{R}^3 \to \mathbb{R}^2$ be the projection of the user's view. Usually, this projection is expressed in the form $Proj(X) = \frac{P \cdot X}{[P \cdot X]_w}$ with P an arbitrary projection matrix. In this case, we derive the following Jacobian:
>
> $$J_{Proj}(X) = \frac{1}{[P \cdot X]_w} \left(P - Proj(X) \cdot P_{w,\cdot}\right)$$
>
> where $P_{w,\cdot}$ is the row of P corresponding to the component w and X is a column vector.

Domain boundaries In order to account for the boundaries of the domain Π of allowed hyper-parameters, we use the active-set method shown in Algorithm 4, inspired from the Prioritized Inverse Kinematics presented by Baerlocher and Boulic (1998). We iterate resolution steps and projections onto the domain, and *freeze* hyper-parameters affected by the projection to their clamped values for the remaining steps. Freezing is done by setting the corresponding column of the jacobian to zero. To avoid breaking the continuity of the solution, we add to the `IsOutOfBounds` test of Algorithm 4 a maximum distance to the hyper-parameter update that was returned at the previous execution of the function (i.e., for $\Delta t = t_{T-1} - t_0$). We also initialize $\Delta\boldsymbol{\pi}$ to the previously returned solution.

We are thus able to handle a point-wise constraint and fulfill the requirements listed above. We see in the next section how we combine multiple such constraints over the extent of the brush.

ALGORITHM 4: Our solver uses an active-set method to account for hyper-parameter boundaries. `Diag(active_set)` returns a diagonal matrix whose j-th coefficient is 1 iff $j \in$ `active_set` in order to *freeze* hyper-parameters that are no longer in the active set.

Input: Jacobian matrix J, target move Δt
Output: An update $\Delta\boldsymbol{\pi}$ of the hyper-parameters
active_set $\leftarrow \{0, \ldots, n-1\}$;
$\Delta\boldsymbol{\pi} \leftarrow (0, \ldots, 0)$;
repeat
  $J^+ \leftarrow$ `PseudoInverse(`$J \cdot$ `Diag(active_set))`;
  $\delta\boldsymbol{\pi} \leftarrow J^+ \cdot (\Delta t - J \cdot \Delta\boldsymbol{\pi})$;
  $\Delta\boldsymbol{\pi} \leftarrow \Delta\boldsymbol{\pi} + \delta\boldsymbol{\pi}$;
  for $j \in$ active_set **do**
    if `IsOutOfBounds(`$\Delta\boldsymbol{\pi}_j$`)` **then**
      active_set $\leftarrow$ active_set $\setminus \{j\}$;
      $\Delta\boldsymbol{\pi}_j \leftarrow$ `Clamp(`$\Delta\boldsymbol{\pi}_j$`)`;
    end
  end
until $\delta\boldsymbol{\pi}$ *is null*;

III.4.2.2 Jacobian buffer filtering

The variations of a single point may not be representative of those of the patch of surface surrounding it, so we sample multiple points within the extent of the brush and average their jacobians. This is still fast because the bottleneck is the evaluation of the parametric shape which is common to all samples (see Section III.4.3.1).

The second motivation for filtering the jacobian buffer is that the L_2 norm, minimized above, is not the most appropriate way to model sparsity. Indeed, we rather need to limit the number of hyper-parameters that have a non-zero update i.e., the L_0 norm. For instance, when two hyper-parameters have a similar influence over the dragged points, we want to use only one of them rather than applying a small change to both.

Hence we refine the user intent with the following model: **(i)** All other things being equal, we want to foster hyper-parameters that show less variation within the extent of the brush. And **(ii)** we want to favor hyper-parameters whose influence over the dragged area would change notably if the brush radius would be increased. Intuitively, this corresponds to making the assumption that the user chooses the maximal brush radius fitting their intent, as illustrated in the drawer example in Figure III.23. We inject extra knowledge about the use case by setting some columns of J_i to zero, thus ignoring the influence of the hyper-parameter over the i-th point.

For objective (i), we compare the coefficients of variation v_k (standard deviation over mean) of the norms of the columns of the J_i within the brush. We discard

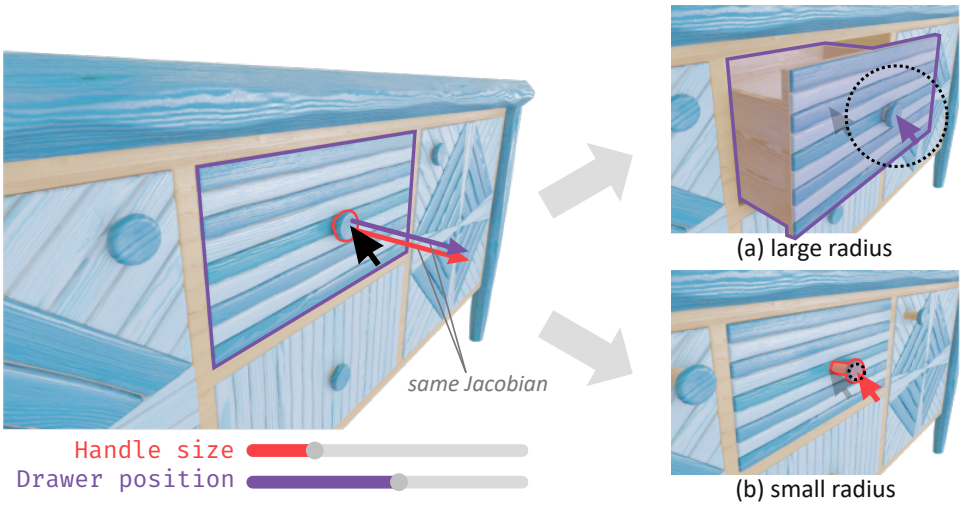


Figure III.23: *Both hyper-parameters of this scene have the very same influence on the drawer's handle. Yet our jacobian buffer filtering enables to distinguish the intent behind the choice of a large (a) or small (b) brush (the dotted circle).*

hyper-parameters such that $\frac{\min v_{k'}}{v_k}$ is lower than a threshold $\lambda_v \in [0, 1]$.

Among the remaining hyper-parameters, we address (ii) by measuring a contrast factor c_k which is the ratio of the average norm of the k-th column of J_i inside of the brush over the one outside of the brush. We foster hyper-parameters that have a high contrast factor, so if $\frac{c_k}{\max c_{k'}}$ is lower than a threshold $\lambda_c \in [0, 1]$, the k-th hyper-parameter is discarded.

Thus, a larger brush is more likely to affect hyper-parameters whose influence has lower frequencies and a pickier brush will affect hyper-parameters with faster variations in the Jacobian buffer. The thresholds translate a global trade-off between L_2 and L_0 sparsities, which would depend on the kind of object that is manipulated. Empirically, L_2 is more important for organic shapes while L_0 is more critical for mechanic ones. In practice, we use $\lambda_v = 0.2$ and $\lambda_c = 0.75$. A high value for λ_c favors sparsity in the modified hyper-parameters, while a high value for λ_v favors regularity in the hyper-parameter selection, i.e., ignoring noisy hyper-parameters.

Single Direction At an extreme edge of this trade-off, we add the possibility to keep only one hyper-parameter. We consider that the beginning of the stroke is more meaningful than the end, because the jacobian information that we have is only valid for small variations of $\boldsymbol{\pi}$, so we pick the one hyper-parameter based on the direction at the beginning of the stroke only, $\Delta t = t_1 - t_0$. For each column j'_k, we look at the cosine similarity (c_{sim}) between j'_k and Δt, as well as the norm $\|j'_k\|_2$. We favor columns with high norm in order to reduce the L_2 norm of the output $\Delta\boldsymbol{\pi}$. On another hand, the higher the cosine similarity, the more exact the

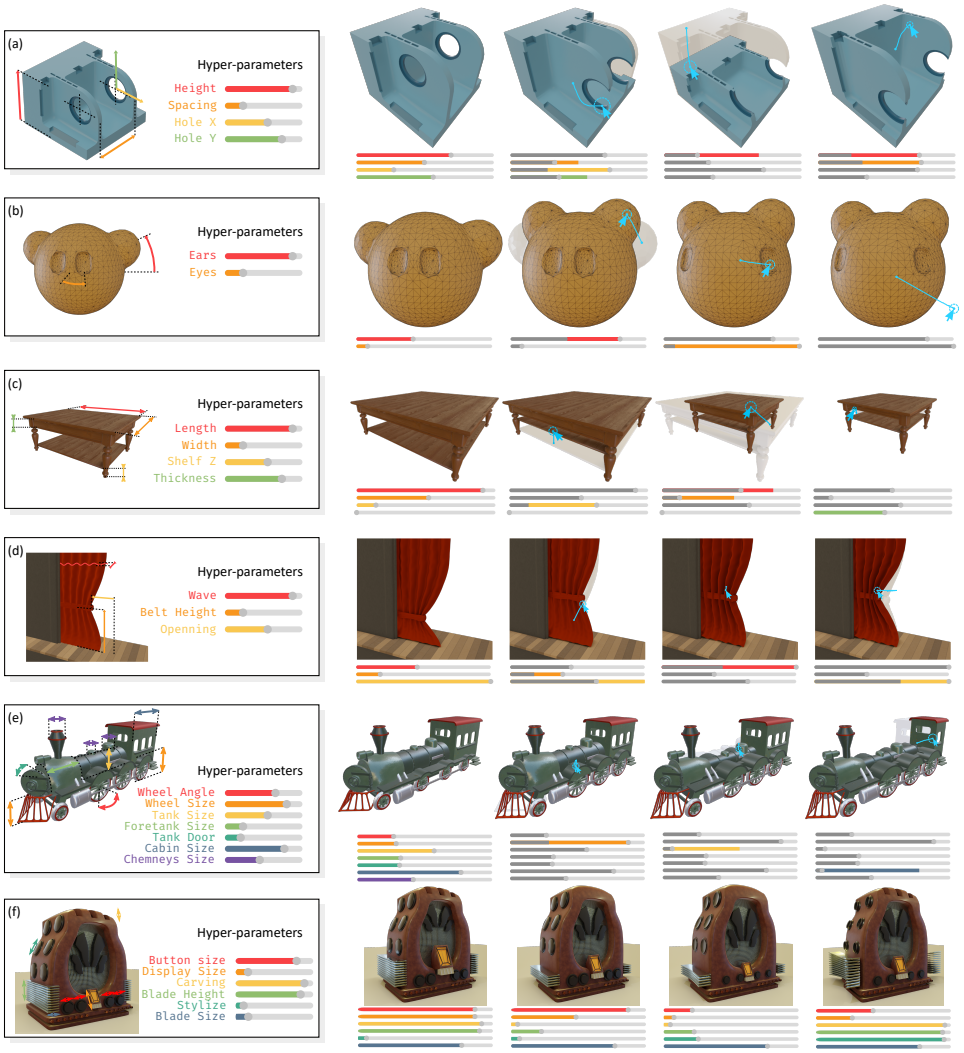


Figure III.24: *Examples of sequences of edits using our method on various scenes. Corresponding DAG amendments can be found in the supplementary material.*

solution. Hence we pick hyper-parameter $\tilde{k}$ based on:

$$\tilde{k} = \underset{k}{\operatorname{argmin}}\, c_{\text{sim}}(j'_k, \Delta t) + \lambda \cdot \left\| j'_k \right\|_2$$

with $\lambda = 1/2$ in practice.

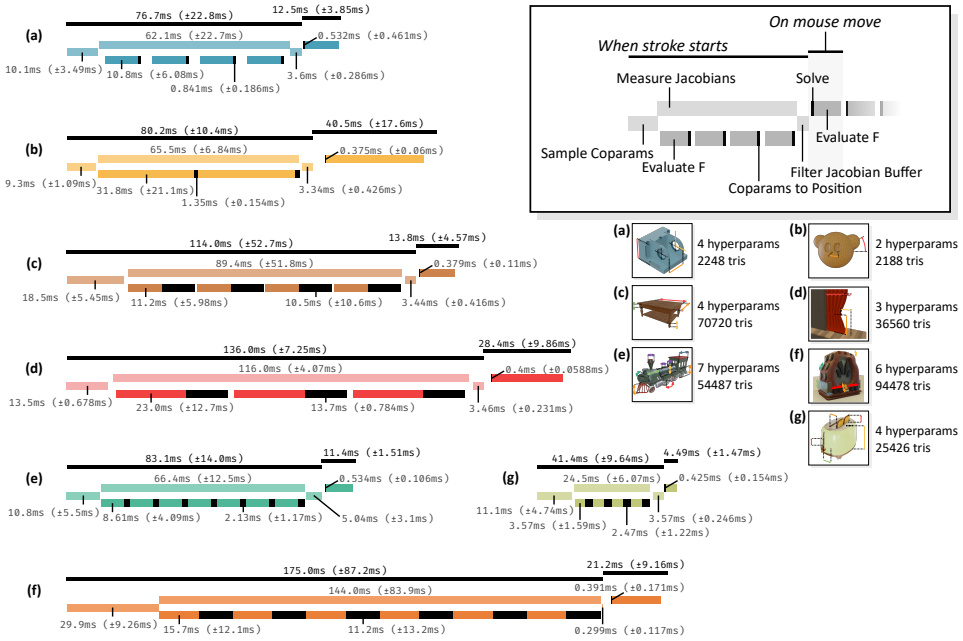


Figure III.25: *Detailed profiling breakdown on several example scenes with varying complexity of DAG and output geometry. All examples are given for 64 sample points. The time needed to evaluate $\mathcal{F}$ does not depend on our method but on the parametric shape engine that we have built onto, and its standard deviation is due to caching mechanisms.*

III.4.3 Results

We implemented our method as an add-on for the *Blender* open source program. Its direct manipulation capabilities are illustrated on a few examples in Figure III.24. In particular, we can observe that examples (a) and (b) exhibit changes of connectivity while the last edit in example (b) shows that clicking in an area not affected by any hyper-parameter induces, as expected, a null update.

III.4.3.1 Performances

For all the examples illustrating this section, the execution time of the DAG amendment is negligible, boiling down to a few milliseconds each time the graph topology is updated. Hence, we focus here on the runtime performance of our system during interaction.

Figure III.25 gives execution time measurements on five scenes. The bulk of the computation is located at the beginning of the brush stroke since the finite differences require many evaluations of the input parametric shape $\mathcal{F}$. The overhead introduced by the solver is negligible compared with the time required to evaluate

$\mathcal{F}$, which is needed anyway to display the current state of the parametric shape.

The overall Jacobian evaluation time is only indirectly related to the number of vertices in the geometry and rather depends on the complexity of the DAG and its nodes' logic. The time needed to retrieve the position of the points from their co-parameters depends on the number of vertices, but since they are grouped by path index l the relation is not strictly proportional. For instance, the table in example (c) has twice as much vertices as the curtain in example (d), but this complexity is mostly concentrated in the legs. The average position evaluation time is 11.3ms, lower than for the curtains, but it has a much wider standard deviation. It peaks around 27ms when points are sampled on the legs but goes below 1ms when dragging elements of the plate.

Performances were measured with 64 sampled jacobians. This count linearly affects the initial sampling of co-parameters, the evaluation of positions from their co-parameters and the filtering of the jacobian buffer. Other elements are not modified. Empirically 64 is a high number of samples in the sense that the output Jacobian is already robust enough for an intuitive interaction at lower values. In practice we use 32 samples, which was way enough for all our examples.

III.4.3.2 Ablation study

To assess the symbiosis of the elements composing our approach, we study here the influence of three of them over the whole system: sample discarding, outbound sampling and path indexing.

Figure III.26 illustrates the importance of discarding sample points after unprojection. Even if they are close to the center of the brush in screen space, the drawers are not on the same plane than the likely area of focus of the user so they should not get affected by the stroke.

In the absence of samples outside of the brush (Section III.4.2.2), the only way to change the size of the handle in the drawer example of Figure III.23 would be to first change the drawer position all the way to its boundary then change the handle and finally move the drawer back to the desired location. Our method makes this same change possible in a single stroke.

Path indices generated by our DAG amendment ensure that there is not two points with the same co-parameter in the output geometry. Without so, if p_i and p_j share the same co-parameter, there is a risk that a row of the jacobian is set to $p'_j - p_i$ instead of $p'_i - p_j$, where p' is the new location of the point p after a slight change of an hyper-parameter. This leads to jacobians totally unrepresentative of the influence of hyper-parameters.

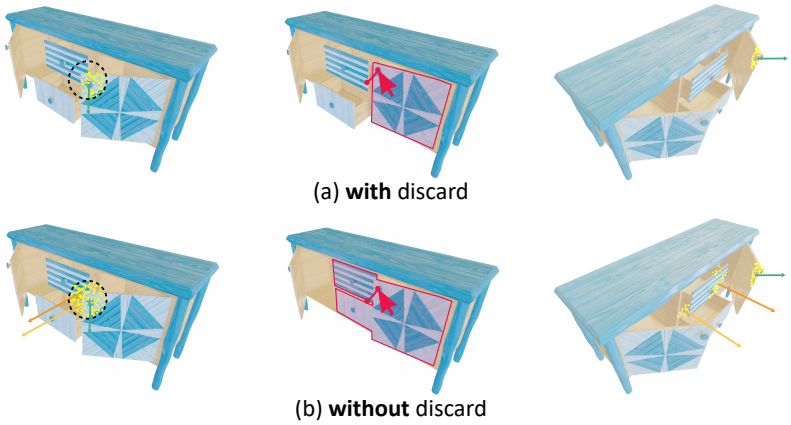


Figure III.26: *Interaction is better localized when we discard samples far from the center of the brush once unprojected in world space (a) than when keeping all points (b). Middle column shows the consequences of a stroke. Right-hand column shows the same interaction under another viewpoint.*

III.4.4 Discussion

III.4.4.1 Properties

As it stands, our method allows intuitive interaction with a parametric shape directly in the 3D view. In particular, a single mouse event can yield multiple hyper-parameters to be updated concurrently. The same parametric shape may also be exposed with various alternative control spaces easily, by simply masking/exposing a subset of its hyper-parameters, making it easy to "publish" the shape for various application scenarios. Moreover, our DAG amendment is non intrusive since we only insert new nodes.

Our approach opens the possibility to apply the many works that have been carried out on IK to parametric shapes that are generated by complex graphs including operations that drastically affect a mesh connectivity (e.g., boolean operations). Not only do we give sense to the notion of Jacobian of a point of the surface but also we propose a filtering scheme to adapt their raw value to the needs of intuitive direct manipulation.

VR ready Our approach is agnostic of the dimension of the interaction space. We have focused mainly on screen based interaction, but any other input device such as VR handles could be used as well. In this case, the projection of manipulation-space sample points onto the geometry at Step 1 of the interaction loop becomes a nearest neighbor search rather than a ray casting.

Implementation Guidelines To integrate our method to an existing shape engine, the latter must expose a way to insert a non destructive operation on

texture coordinates before/after existing operations. The implementation must list for each available operation the number of duplicates it may create and a mean to retrieve the duplicate index j. The interaction loop expects that the host software provides the user input, a way to query the geometry attributes at sample points on the screen and a way to evaluate the DAG programmatically.

User Feedback We presented the tool to 19 users whose proficiency with 3D software ranges from absolute beginner to professional, asked them to reach a target configuration of the parametric shape, then collected their feedback on scales from 1 to 5. Users were able to manipulate almost all the hyper-parameters they wanted (only 1/4 felt blocked and it was at most on a single hyper-parameter) and felt confortable with completing the task (63% found it rather easy). In the majority of cases (63%), they used our brush exclusively or felt back only a few times to the sliders (resp. 42% and 21%). Professional users, used to hand-crafted manipulators, were sometimes frustrated not to be able to target for certain a given hyper-parameter, but we recall that such manipulators require extra work when originally creating the parametric shape, which our method does not. On average, users were leaning towards our brush rather than the sliders and would be likely to use it in their usual 3D software. More extensive results are available in the supplementary material of É. Michel and Boubekeur (2021a).

III.4.4.2 Limitations

Homogeneity Measuring the norm of an hyper-parameter update $\Delta\boldsymbol{\pi}$ is ill-defined because hyper-parameters are in general not homogeneous to each other, namely they are expressed in different units. This is why our Jacobian buffer filtering takes care of only comparing affine invariant properties (coefficient of variation, contrast factor), but it remains a problem to properly define the objective of sparsity of $\Delta\boldsymbol{\pi}$ in the presence of diverse units.

First order We currently only measure first order information about the parametric shape – the Jacobians – and do it only once, at the beginning of the stroke. For long strokes, hyper-parameters that have a non linear behavior are thus incorrectly interpreted. Furthermore, when the evaluation time of $\mathcal{F}$ increases, the delay needed to compute the jacobians starts to be noticeable, between the click and the first update of the hyper-parameters.

III.4.4.3 Future prospects

Global sampling We could try to precompute jacobians before the beginning of the stroke – while the user is changing the view point for instance – to avoid the slight lag when the interaction begins. This might require to use an acceleration structure to find the nearest neighbor of w in the new geometries as there would

be more sample points to consider, or would require to store the cooked geometries G', costing memory.

Other type of input constraints In our model, the user constraint takes the form of a brush stroke, but the Jacobian information we derive can be used with other types of input: layout-based constraint, image-based input (potentially coming from a differentiable renderer), global metrics (e.g., conserving volume) or visibility-based constraint (camera or illumination). This can potentially lead to interesting novel workflows.

Follow-ups After our original publication of this work (É. Michel & Boubekeur, 2021a), two papers explored the similar issue of directly manipulating shape programs. Gaillard et al. (2022) uses box proxies to speed things up, in the case of part-based models, and developed an advanced solver with solution clustering. Cascaval et al. (2022) built their own automatic differentiation system and reach a much more scalable result in terms of number of hyper-parameters. They also use global energies to disambiguate the user intent. Both of these work are restrained to constant connectivity though, and notice that a better integration with ours could be beneficial.

III.4.5 Conclusion

Our method leverages the information provided by the parametric shapes when seen as programs – described in general as graphs of operations – to make inverse control available to them in an intuitive brush-based interaction loop. Our approach may pave the way for more advanced uses of graph-based shape representations, exploring our local differentiation scheme with alternative optimization strategies.

IV

Tiles-based declarative programming of shapes

IV.1 Introduction

IV.1.1 Constrained layout

As we highlighted in Chapter I, imperative DAGs are not the only paradigm for shape programming. Shape programming is about identifying systematic behavior in the creation process and modeling this systematism. DAGs represent rules applying to the flow of geometry processing operations, but in this chapter we represent rules that apply to the end result, to the layout of pieces of geometry.

There are two main families of constrained layout: free-position layout, and tiled layout. The former is for instance about ensuring alignments, contacts, regular distribution, orientations, etc. The latter is quite different: positions and orientation of the *slots* where pieces of geometry must be instantiated are pre-determined, but the unknown is which piece – which *tile* – goes where.

In both cases, the power of using shape programming at the level of declarative layout constraint is that it naturally enables one to combine manually authored fixed content with shape programming. However, in the case of a tiled layout, a lot of constraints apply to the content of the tile itself. Indeed, the geometry of tile edges – *interfaces* – that are in contact in the final layout must match.

Although the problem of laying out tiles given their neighboring constraints has been studied a lot, authoring the geometric content of the tile remains quite cumbersome. A striking example of how this authoring can be time consuming is the tiles that Stålberg presents in their break down of *Townscaper* (Stalberg, 2018), where they modeled hundreds of tiles while ensuring these constraints manually

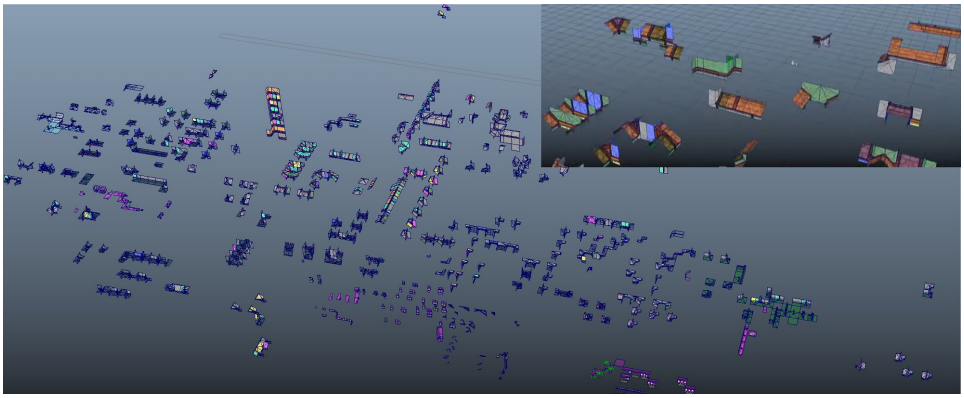


Figure IV.1: *Some of the ca. 500 tiles needed by the* Townscaper *tile-based modeling game, manually modeled using* Autodesk Maya. Courtesy of O. Stålberg (https://www.youtube.com/watch?v=1hqt8JkYRdI).

(Figure IV.1).

This chapter focuses on easing the use of tiled layout by assisting the construction of rich tile sets and in particular of the 3D content of their tiles. We first present an approach based on growing geometry from 2D cross-sections drawn on the interfaces between tiles (Section IV.2). We show how this integrate into a system for quickly authoring auto-similar mesostructures. We then explore how to bring diversity in the content of tiles by replacing their static geometry with parametric shapes (Section IV.3). The tile space becomes continuous and we have to solve for both neighboring constraints and the hyper-parameters driving the content.

IV.1.2 Wang Tiles

IV.1.2.1 Definition

Prior to detailing our contributions, let us first define the framework of *Wang tiles* (H. Wang, 1961), which is at the heart of most tiled layout system. First, a problem of Wang tiling is in particular a tiling problem, meaning that we have a set of geometric tiles T that must be laid out on a domain with *no overlap* and *no spacing* between tiles. This is what makes a tiled layout different from other kinds of object layout. Each tile from T may be instantiated multiple times to solve this problem.

In general, tiling can lead to very complex setups, like for instance Penrose tiles (Penrose, 1974), and has fascinated mathematicians and physicists for a long time. This complexity is the reason why it was attractive to the graphics community, for content generation, but it makes it really hard to find a valid layout of tiles. Wang tiles get rid of one part of the difficulty of tiling by pre-defining tile locations. There remains the question of which tile to instantiate at each location,

Figure IV.2: *Wang tiles are tiles for which adjacency rules are given by labelling the edges of each tile. Two tiles can be neighbors only if the edges thus put in contact are labelled with the same color.*

but this becomes a purely discrete problem.

Originally, locations are predefined by considering only square tiles, so that they are distributed on the vertices of a regular grid, which we call *slots*. We see in Section IV.2 that this can be more flexible when allowing tiles to be deformed to fit their slot. We use the term *generalized* Wang tiles in our formalism to stress out the fact that they are not especially squares.

Some tilling problems only rely on the shape of the tiles to state which tile fits next to which other one, a bit like a puzzle does. On the contrary, adjacency rules for Wang tiles are given by labeling the edges – a.k.a. *interfaces* – of each tile (Figure IV.2).

To summarize more formally, a generalized Wang tile set is given by a tuple $\mathcal{T} = (T, I, D, L)$. T and I are a discrete set of abstract symbols representing respectively tiles and interface labels. D is a set of directions, to designate tile edges. And $L : T \times D \rightarrow I$ is the labeling function, telling the color $i \in I$ of the edge in direction $d \in D$ in the tile $t \in T$. In the case of square tiles, we use for instance $D = \{\mathcal{N}, \mathcal{S}, \mathcal{E}, \mathcal{W}\}$ for *north*, *south*, *east* and *west*. When used in computer graphics, tiles come with a visual content (image, mesh, etc.) that is instantiated at each slot where the tile is used.

IV.1.2.2 Related Work

Tile-based content generation In computer graphics, tiling algorithms were first applied to procedural texture generation. The *aperiodic texture mapping* of Stam (1997) shows how laying out multiple patches of texture can efficiently break the visual repetitivity that strikes the human eye when naively repeating the same image. Following Stam's, multiple other papers explored tile-based graphics generation. Neyret and Cani (1999) used tiles for on-surface synthesis rather than for paving a plane, thus performing seamless texturing. They took some liberties with the original Wang tile framework, first changing the tile shape to triangles, but more importantly introducing the need to orientate tile edge labels, which we also experienced in our method.

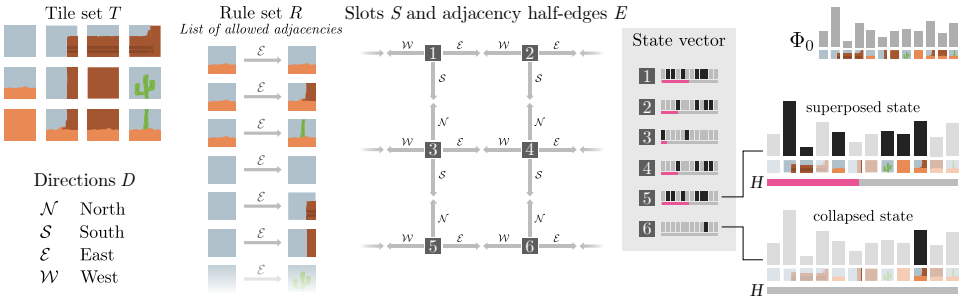


Figure IV.3: *To solve a Wang tiling problem, the* Wave Function Collapse *algorithm maintains for each slot a superposition of all tiles that may be assigned and progressively reduces this set, starting with slots of lower entropy H.*

An important requirement for applying tile-based generation is to define a *tile set* the generator will draw from. For texture generation, Cohen et al. (2003) extract this set by stitching patches of an input example. Compared to other texture synthesis techniques such as *image quilting* (Efros & Freeman, 2001), tile-based texture generation limits the computational workload involved in blending texture patches: once the graph cut sewing is prepared for each tile, the synthesis itself is very fast – it only consists in laying out tiles – and can be done on the fly during real time rendering. Similarly to *image quilting*, the *mesh quilting* method (K. Zhou et al., 2006) does not benefit from such a computing factorization.

In their representation of forest scenes, Decaudin and Neyret (2004) present tiling as a mean to compactly encode the geometry of all trees, since they precompute light transport only for a set of tiles before instantiating them on the fly at render time. We follow a similar spirit in our mesostructure representation, using tiles to share memory.

Tiling engines The tiling engine is responsible for finding a valid layout of tiles, given a tile set and a domain of slots to cover. Stam (1997) uses a predefined tile set for which a constructive algorithm for aperiodic tiling is known to always work (Grünbaum & Shephard, 1987). Neyret and Cani (1999) consider the exhaustive tile set where all combinations of Wang labels are available, thus tiling is always solvable. The approach of Cohen et al. (2003) is more flexible than Stam's, as the tile sets are generated depending on the number of Wang labels such that they can use a tiling algorithm that always succeeds.

In all of these approaches, the tile set itself is still an internal entity the user does not have direct control on. In the approach we present in Section IV.2, the tile set is exposed as the primary lever for design, so we had to turn to more generic tiling engines, based on *possibility space collapsing* (Merrell, 2007; Gumin, 2016) (Figure IV.3).

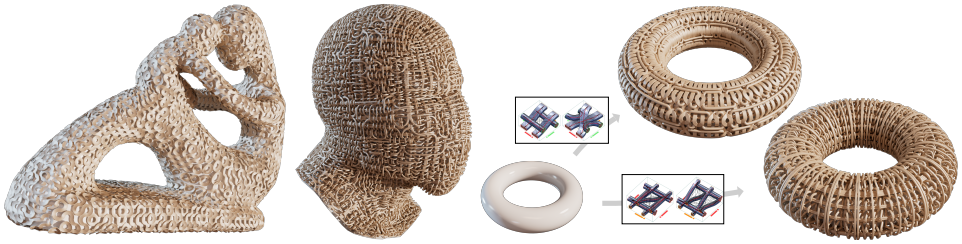


Figure IV.4: *Mesostructures produced using our interactive design tool in a few minutes. Our method enables a quick design of very intricate topology, and as illustrated on the torus the same input macrosurface can lead to various styles depending on the tiles created through our proposed workflow.*

IV.2 Tile-based geometric amplification

IV.2.1 Problem Setting

IV.2.1.1 Geometric amplification

Geometric surface enrichment is often achieved using displacement mapping, for which content can easily be authored using standard (2D) painting tools. However, the content injected onto the macrosurface has fixed disk topology and cannot represent complex structures, such as tunnels and handles. To overcome this issue, generalized displacement mapping and shell mapping are explicitly modeling the surrounding space of the macrosurface and use various mechanisms to instantiate complex shapes in it. Unfortunately, this comes at the cost of tedious authoring, as the mesostructure shall still behave like a mappable object, conforming to tilability constraints and deforming following the macrosurface curvature. As a consequence, only complex preprocessings (K. Zhou et al., 2006) acting on preexisting geometry have been developed so far to transform a 3D surface into a proper mesostructure.

We propose a new approach to self-similar mesostructure design built upon Wang tiling and adopting an interface-centric workflow, where the user creates mesostructure atoms through the 2D cross-sections they form at tiles interfaces. Our approach can be executed on any quad-based surface domain and runs in real time, allowing the user to quickly create complex mesostructures in a few brush strokes. Just like displacement maps, our resulting model can be reused across macrosurfaces – with minimal tile set adjustments – and as we will see in Section V.2 is architectured to be compact and GPU-friendly.

Contributions Our main contributions are: **(i)** a mesostructure design workflow centered on continuity by construction and built upon a tiling engine, **(ii)** a tiling engine, evolving state-of-the-art with user-prescribed constraints, and **(iii)** a mapping mechanism leveraging the procedural nature of our mesostructure model

to populate the macrosurface shell space.

IV.2.1.2 Related Work

Enriching a coarse surface with sub-polygonal details has been a subject of interest quite since mesh-based representations are used. These details are typically mapped like textures; the first example is displacement mapping (Cook, 1984), which deforms polygons along their normal, and then came iterations like *View-Dependent Displacement Mapping* (L. Wang et al., 2003) and *Generalized Displacement Maps* (X. Wang et al., 2004). But as more expressivity is provided to the map-based geometry, it becomes unclear which part of the geometry should be encoded in the original mesh and which part belongs to the displacement map, in order to keep good rendering performance. On another hand, displacement-based methods are still constrained to the topology of the original macrosurface, so *shell maps* (Porumbescu et al., 2005) were proposed, using surface meshes as 3D texture data of arbitrary topology along a macrosurface. This work led to a number follow-ups, adapting it for real-time mapping (Ritsche, 2006), mitigating deformation artefacts (Jeschke et al., 2007) or using it for geometry transfer (Takayama et al., 2011). In between lies hybrid approaches like *relief mapping* of complex mesostructure topology (Policarpo & Oliveira, 2006) (but limited to a few overhanging layers). A more radically different approach to geometric texture mapping is to leverage an implicit representation of the macrosurface (Brodersen et al., 2008). Our approach makes no exception to the overall surface amplification scheme: the meso-scale geometric content is defined in a few unit cubes – the tiles – and then mapped onto the target surface.

Empirically, a limiting factor when using tile-based modeling is the creation of tile content that remains seamless at any time. Many approaches are data-based, taking an example as input (Bhat et al., 2004; Lagae et al., 2005; K. Zhou et al., 2006; Merrell, 2007; Gumin, 2016). Although this works well for 2D raster images, it is much harder to define in the case of 3D vector content laid out on irregular grids (Merrell & Manocha, 2008), so in practice tile based 3D mesh generation uses manually crafted atoms. For 2D vector tiles, Brian et al (2018) propose an editor in which, while drawing on tiles, the user sees an onion skin of the continuation lines of neighboring tiles. Porting this approach to 3D content is not straightforward, and our work draws from this spirit of attributing a predominant role to interfaces during authoring. When not based on arbitrary examples, detail generation methods can also be domain-specific (Landreneau & Schaefer, 2010). De Toledo et al. (2008) provides a comparison of various mesostructure techniques and Koniaris et al. (2014) reviews more specifically volumetric texture mapping.

IV.2.2 Method

Our workflow is presented in Section IV.2.2.1) and summarized in Figure IV.5. It is based on a factorized, highly structured representation of the mesostructure

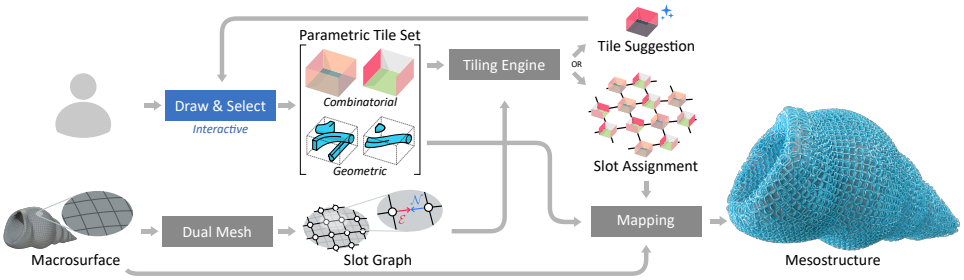


Figure IV.5: *Overview. Draw & Select step is the step involving user interaction; it is detailed in Figure IV.6. Other steps are autonomous.*

(Section IV.2.2.2) which feeds a tiling engine exposing a feedback loop to the user for efficient authoring (Section IV.2.2.3) and for which we propose dedicated mapping (Section IV.2.2.4). The interoperability of our approach with real-time rendering will be detailed later in Section V.2.

IV.2.2.1 Design workflow

Our method takes a mesh representing the macrosurface as input, along which the mesostructure is to be generated. Basically, the user designs the mesostructure by creating progressively a set of tiles, while a tiling engine cover the macrosurface by instanciating consistently and rendering a tile arrangement on-the-fly (Fig. IV.6).

In the tile set, a tile is defined by **(i)** a geometric content and **(ii)** adjacency rules, with the geometric content being instanced each time the tile is used by the tiling engine. Following Wang tiles, adjacency rules are specified by labeling the four sides of a tile. Similarly to Neyret and Cani (1999), we also add an orientation flag to these interface labels, and only interfaces that are a mirror of each other may be juxtaposed.

The main friction when defining the content of a tile is to ensure that it is consistent with the content of any other tile that the tiling engine could place next to it. This is why we take the problem the other way around: in our approach, users author geometric content by drawing 2D cross-sections on the tile's interfaces. As such, the tile's geometric content is entirely defined by **(i)** assigning interfaces to the four sides of the tile and **(ii)** selecting pairs of cross-sections to connect using a *sweep surface*. The continuity of the mesostructure across interfaces is thus ensured by construction.

Another source of friction in the creative process relates to tiling engine failures. Since we let the user design arbitrary tile sets, and since the tiling problem is NP-hard in general, this happens on a regular basis, even with the best in class tiling engine. Consequently, we designed our tiling engine to suggest the addition of a new tile to the user whenever it gets stuck, specifying which configuration of interfaces could have enabled it to pave the whole macrosurface in an interactive

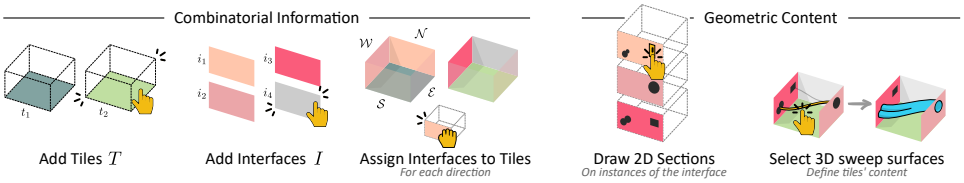


Figure IV.6: *Tile set authoring interactions involves setting combinatorial information (tile interfaces) and geometric content (2D cross-sections and 3D sweep surfaces).*

feedback loop. To the best of our knowledge, there is no prior example of such a joint design of a tile set.

IV.2.2.2 Procedural mesostructure model

Our mesostructure model compactly represents its surface elements in a factored way. Essentially, it takes the form of a tuple (T, I, M, A) composed of a tile set (T, I), a macrosurface M and an assignment A of tiles to the macrosurface, as summarized in Figure IV.5.

Tiles The *tile set* is formed by **(i)** a set I of interfaces containing 2D cross-sections, as well as **(ii)** a list T of tiles. A tile contains for each of its four sides a reference to one of the interfaces, as well as a flipping flag: we note $\overleftrightarrow{i_k}$ the flipped version of an interface i_k. A tile also contains a geometric content, given as a list of 3D sweep surfaces, each referencing a pair of 2D cross-sections interpolated along a procedural 3D Bézier curve (Figure IV.7). Additional 3D content may be injected into the tile, provided it is entirely contained within the extent of the tile, i.e., it does not interact with the interfaces. Lastly, a tile contains a set of flags indicating whether the tiling engine is allowed to flip and rotate it.

Interfaces The 2D content of each interface – instantiated on each tile side that references this interface – is modelled as a binary space occupancy function over the unit square. During the design phase, connected components are dynamically detected and constitute the cross-sections that can be selected for generating sweep surfaces.

The *macrosurface* M is the domain where tiles are instantiated, extending the usual case of grid generation. It takes the form of a quad surface mesh and defines the associated *slot graph* (S, E), namely the undirected dual graph of the quad mesh connectivity where a vertex s of the slot graph (a *slot*) corresponds to a face of the quad mesh. Each half-edge $(s, e) \in S \times E$ of the slot graph is labelled with a direction $d \in D$, with at most one use of a given direction per slot. This indicates how a tile should be instantiated on this slot. Vertex positions and normal vectors define the shell space (Porumbescu et al., 2005) in which the mesostructure lives.

The *slot assignment* $A : S \longrightarrow T \times P$ provides for each slot a *transformed tile*, namely

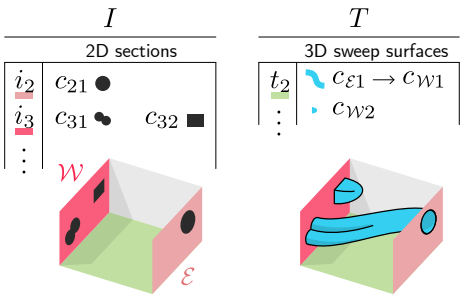


Figure IV.7: *The geometric content of tiles is defined by sweeping across 2D cross-sections drawn on interfaces. The same cross-section may be used by more than one sweep, and if they are not used by any, a* cap *surface is automatically added.*

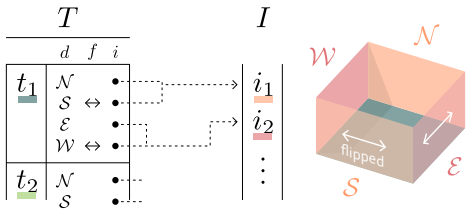


Figure IV.8: *Wang labelling: A tile references, for each direction $d \in D$, one of the tile interfaces $i \in I$ whose geometric content is constrained to comply with. The interface can be horizontally flipped.*

a tile index and a tile transform indicating whether the tile should be rotated and/or flipped. This transform takes the form of a permutation $p \in P$ of its four base corners. The tiling engine decides for each slot which transformed tile it assigns, ensuring that the interface assigned to an half-edge is always the flipped version of the interface of its opposite half-edge.

IV.2.2.3 Tiling

Constraint solving Given the tile set and the slot graph, the tiling engine assigns a tile and its transform (rotation/flip) to each slot, such that neighboring tiles always have matching interfaces.

Our tiling engine is largely based on the *Wave Function Collapse* (WFC) algorithm (Gumin, 2016), itself following mostly the engine proposed by Merrell (2007). It proceeds by progressive reduction of the possibility space, alternating two steps. Initially, the set of all tiles is assigned to each of the slots, then it greedily propagates constraints through a depth-first traversal of the slot graph. Each time this recursive propagation (*collapse*) step reaches a fixed point, the possibility set of one of the slots is arbitrarily reduced to a single tile (*observe* step). In order to reduce the chance of leading to a dead-end – a case where the possibility set of a slot is empty – the observed slot is chosen so as to minimize the amount of informational entropy removed from the system. In the case of equiprobable tiles, this simply means we observe the slot with the smallest possibility set (that has more than 1 tile). When stuck, the algorithm restarts with a different random seed.

The tiling problem being NP-hard, this algorithm does not magically handles all cases, but benefits from some nice properties. First, it is easy to implement, and has proven to be useful in practice, especially for video games (Stalberg, 2018).

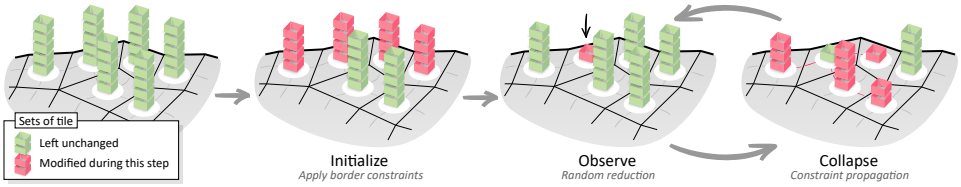


Figure IV.9: *Outline of the tiling solver described in Algorithm 5. The possibility space of each slot is initialized to the set of all tiles, then our border exempt/only interfaces imposes some initial constraint, and the remainder of the algorithm is an alternation of arbitrary local choices and depth-first constraint propagation.*

Secondly, it is not tied to the regular grid structure on which tiling algorithms are usually applied; we were able to adapt it to the arbitrary slot graph derived from our input macrosurface with minimal modification. Lastly, reasoning about possibility spaces is a flexible framework in which it is easy to encode extra constraints, like forcing some interfaces to occur only on the boundaries of the macrosurface. Other work even ensure path finding or other non-local constraints (Sandhu et al., 2019), and these could be ported to our use case.

Non regular slot graph To adapt the tiling engine to an arbitrary slot graph, each half edge of this graph is labeled with a *direction* $d \in D$. For each slot, we set at most one of the half edges per direction. A slot has never more than four half-edges since it is the dual of a quad mesh, and it can have less when the quad lies on the boundary of the macrosurface. When the slot graph is a regular grid, a half-edge labelled with a *north* direction will always face a half-edge labelled with a *south* direction, but for an arbitrary graph, it is not necessarily the case (which is why we reason based on half-edges).

The sole constraint ensured by the solver is thus the following: let s_1 and s_2 be two slots that are connected by an edge e in the slot graph, and d_1 (resp. d_2) the direction labeling the half-edge (s_1, e) (resp. (s_2, e)). Two (transformed) tiles t_1 and t_2 can be assigned to s_1 and s_2 only if the interface attached to the tile t_1 in direction d_1 is the same as the one attached to the tile t_2 in direction d_2 but horizontally flipped.

Boundary constraints The user may annotate tile interfaces with two flags: *boundary exempt* and *boundary only*. The first one specifies that the interface must never occur in a direction that is connected to no other slot. This is typically used for any non empty interface when the user does not want open ended sweeps. The second flag tells that the interface must never be connected, it is allowed only on boundaries. This can be used to ensure that the generated shape is made of one single piece, without including empty interfaces. These flags do not really interfere with the solving algorithm, they can be fully applied as a preprocessing of the possibility space (Algorithm 6). For each slot that has no half-edge labelled

ALGORITHM 5: Outline of the tiling solver. Pink underlined items show our additions to the typical WFC algorithm (Gumin, 2016): (1) `RecordNeighbors` saves the cause of the dead-end for the tile suggestion mechanism, and (2) we traverse an arbitrary slot graph rather than a regular grid.

Data: Slot graph $G = (S, E)$ and tile set T
Result: Slot assignment $A : S \to \mathcal{P}(T)$
fn Solve G, T**:**
  $A_0 \leftarrow$ InitialConstraints(G, T);
  $A \leftarrow A_0$;
  repeat
    try:
      $s_0 \leftarrow$ Observe(A);
      Collapse(s_0);
    catch *Finished*:
      return A;
    catch *DeadEnd*:
      $A \leftarrow A_0$;
    end
  end
end
fn Observe A**:**
  if *exists* $s \in S$ *such that* $|A[s]| = 0$ **then**
    RecordNeighbors(s) ; (1)
    throw DeadEnd;
  end
  if *not exists* $s \in S$ *such that* $|A[s]| > 1$ **then**
    throw Finished;
  end
  $m \leftarrow \min(|A[s]|$, for slots $s \in S$ such that $|A[s]| > 1)$;
  $s_0 \leftarrow$ random slot such that $|A[s_0]| = m$;
  $A[s_0] \leftarrow \{$ random tile from $A[s_0]\}$;
  return s_0;
end
fn rec Collapse s_1**:**
  foreach *direction* d_1 **do**
    $(s_2, d_2) \leftarrow$ neighbor half-edge of (s_1, d_1) ; (2)
    ResolveConflicts(s_1, d_1, s_2, d_2);
    if s_2 *changed* **then**
      Collapse(s_2);
    end
  end
end

with a given direction d, we initially remove all tiles whose interface in direction d is *boundary exempt*. And for each direction for which there exists a half-edge, we remove tiles whose corresponding interface is *boundary only*. Then we feed the tiling engine with this initial possibility space.

ALGORITHM 6: Initialization of the possibility space prior to running the tiling engine. The pink underlined section shows how border exempt/only interfaces can easily be integrated.

Data: Slot graph $G = (S, E)$ and tile set T
Result: Slot assignment $A : S \to \mathcal{P}(T)$
fn InitialConstraints T**:**
  foreach *slot* $s \in S$ **do**
    $A[s] \leftarrow T$;
    foreach *direction* $d \in D$ **do**
      if s *has* ***no*** *half-edge labelled* d **then**
        $A[s] \leftarrow A[s] - \{t \in T \mid$ the interface of t in direction d is **border exempt** $\}$;
      else
        $A[s] \leftarrow A[s] - \{t \in T \mid$ the interface of t in direction d is **border only** $\}$;
      end
    end
  end
  foreach *slot* $s \in S$ **do**
    Collapse(s);
  end
end

Tile suggestion Our tile suggestion mechanism is executed during the feedback loop between the model and the user, in order to address the fact that tiling can be arbitrarily hard or even not possible for a given tile set on a given macrosurface. When stuck for too long, the tiling engine provides the user with a new tile, specifying the configuration of interfaces that would have helped it.

To do so, we introduce a greedy algorithm based on the following voting scheme (Algorithm 7). We consider the set L of all tile side configurations that can be generated from the set of interfaces I. Each time the possibility set of a slot becomes empty – forcing the tiling engine to backtrack – a vote is cast for all configurations of L that are compatible with the possibility set of its neighbors. All possible transformations (rotation, flip) are applied to a configuration of L when checking that it can fit. For instance, if a tile t in the possibility set of the *north* neighbor shows the interface i in the direction of the empty slot (*south* if we are on a regular grid), then all elements of L labeled with $\underset{\leftrightarrow}{i}$ in the *north* direction receive a vote. The algorithm then suggests the tile that received the highest number of votes.

IV.2.2.4 Shell Mapping

Once a transformed tile is assigned to a slot, the last stage of our framework aims at mapping it to the actual shell space of the macrosurface for rendering. We leverage the natural relationship between a single slot from our graph and

ALGORITHM 7: Our tile suggestion algorithm is based on a voting system. In practice we also label votes with the transform p and break ties in the argmax by maximizing the number of identity transforms.

Data: Dead end neighborhoods N recorded during solving. A neighborhood $n \in N$ gives for each direction $d \in D$ a set of possible transformed interfaces $n_d = \{i_1, \underset{\leftrightarrow}{i_2}, \dots\}$ (where $\underset{\leftrightarrow}{i_2}$ means that interface i_2 is flipped).
Result: An interface i_d for each direction $\overrightarrow{d} \in D$ of the new tile
fn SuggestNewTile N:
  Initialize votes: $I^4 \to \mathbb{N}$ to $\mathbf{0}$;
  foreach *neighborhood* $n \in N$ **do**
    foreach $\boldsymbol{i} \in n_{\mathcal{N}} \times n_{\mathcal{S}} \times n_{\mathcal{E}} \times n_{\mathcal{W}}$ **do**
      foreach *tile transform* p **do**
        $\boldsymbol{i'} \leftarrow \text{inverse}(p) \cdot \boldsymbol{i}$;
        $\text{votes}(\boldsymbol{i'}) \leftarrow \text{votes}(\boldsymbol{i'}) + 1$;
      end
    end
  end
  return argmax(votes);
end

the hexaedron extruded from its corresponding quad on the macrosurface. More precisely, we cast the mapping problem as a deformation one (Porumbescu et al., 2005), from the mesostructure normalized space to the shell one. We express the geometric content of a tile w.r.t. the 8 corners of its slot's bounding box and use these local coordinates to reexpress it w.r.t. the extruded quad, taking inspiration from cage-based deformation.

The *Shell Mapping* approach (Porumbescu et al., 2005) can be reformulated in our case by replacing the barycentric interpolation performed over a tetraedrization of a prism extruded from a triangle with a generalization of barycentric coordinates to non-simplex boundaries – such as *Mean Value Coordinates* (Ju et al., 2005) for instance – computed within the hexaedron extruded from a quad. However, although this yields smoother deformation than dicing the hexaedron in tetraedrons and applying Porumbescu et al's̈cheme, significant distortion still subsists.

To contain it, we use the parametric nature of the tile's geometric content, i.e., sweep objects, and deform their trajectory curves first, before the sweeping step. Doing so, the 2D cross-sections preserve their expected shape, e.g., a circle will produce a perfect tube, not an ellipsis-based one. Note that this may be opted out if one aims at deforming cross-sections as well, but we found empirically that cross-section preservation is often the expected behavior.

As our trajectories are cubic Bézier curves, our mapping problem now boils down the positioning for each of them their four control points $(p_0, ..., p_3)$ in shell space. To ensure tangential continuity of the trajectories across interfaces, p_0 (resp. p_3) is moved to its corresponding interface and expressed using bilinear interpolation

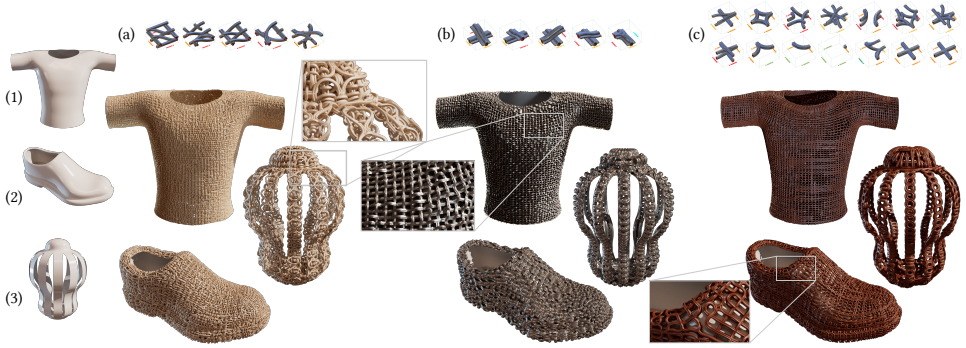


Figure IV.10: *Once designed, the same tile set can be applied to various macrosurfaces.*

over its 4 corners. Meanwhile, its tangent is set along the normal of the corresponding hexahedron face n_0 (resp. n_3), yielding the remaining control points $p_1 = p_0 + m_0 n_0$ (resp. $p_2 = p_3 + m_3 n_3$). The magnitude m_0 (resp. m_3) of these tangents is defined as follows:

$$m_i = m \frac{\alpha_i}{\alpha_i + \alpha_{3-i}} \text{ for } i \in \{0, 3\}$$

with $m = 8d\frac{\sqrt{2}-1}{3}$ and $\alpha_i = \angle(p_{3-i} - p_i, n_i)$. When both p_0 and p_3 are on the same interface, the diameter d is set to $||p_1 - p_2||$. When ends are on neighboring interfaces, this distance is multiplied by $\sqrt{2}/2$ so that the curves approaches a section of circle. When they are on opposite interfaces, we use the same value of d but set the weights α_0 and α_3 to 1 since $\alpha_0 + \alpha_3$ is null. We adopted this heuristic for its visual consistency, and the value of m_0 and m_3 can be globally or locally scaled by the user to produce various looks.

IV.2.3 Results

IV.2.3.1 Experiment

The renderings from Figure IV.4 have been computed using a third party render engine. Our real-time visualization is detailed in Section V.2. Figure IV.10 shows that once a tile dictionary has been defined, it may easily be used across multiple macrosurfaces, applying a similar style to various shapes, while only requiring from the user that they create missing tiles corresponding to unseen topological configurations.

Figure IV.11 illustrates the interest of manipulating a procedural representation of tile content when it comes to mapping the content into a cell of the macrosurface's shell. Rather than blindly deforming the synthesized mesostructure, we deform the input of the procedural construction, namely the control points of the sweep's trajectories. This leads to a more natural deformation, that conserves the aspect ratio of user-drawn 2D cross-sections.

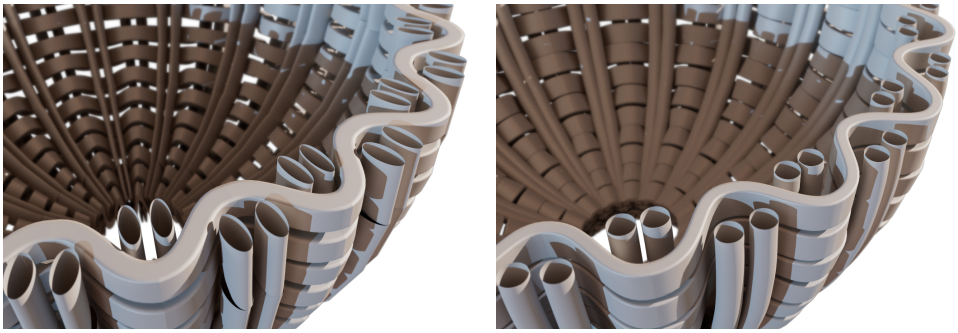


Figure IV.11: *When mapping a tile's content into an hexahedron of the shell, deforming each point of the generated surface (left) leads to more distortion than applying the deformation to the underlying curves, prior to sweeping (right,* ***ours****).*

Figure IV.12: *When the macrosurface has open borders, one can force the tiling to place an empty interface at boundaries to prevent open geometry (middle and right). It is also possible to prevent this empty interface from occurring away from boundaries. (right).*

The basket shown in Figure IV.12 is a typical use case of our border constraints. Without any constraint, open ended surfaces appear on the boundaries of the mesh (Figure IV.12.a). The user can then add an empty interface and flag all the other ones as *border exempt*, so only the empty interface is used at boundaries (fig. IV.12.b), and prevent disconnection using the *border only* flag (fig. IV.12.c).

IV.2.3.2 Discussion

We proposed a method for efficient authoring and representation of rich 3D mesostructures along the surface of a quad mesh. Our approach is purely user-driven, on the contrary to data-driven approaches such as *Mesh Quilting* (K. Zhou et al., 2006) which are less interactive since all a user can do is provide a different input example. An interesting follow-up would be to study how to hybrid one method with the other.

We reduce the boilerplate involved in defining the 3D content of tiles by integrating the constraint of continuity at tile interfaces from the very beginning of the user

interaction. And as a by-product, the parametric nature of the content interacts nicely with the mapping into the shell space, mitigating deformation.

IV.2.3.3 Future Work

Our constructive method could be extended to more general content, applied to a 3D slot graph, although this would imply multiple user interface challenges. Besides, it is quite straightforward to adapt to other polygons than quads, using one tile set per corner count.

To give more control to the user and we could provide the possibility to force a particular tile to be present at a given slot, or to force the regeneration of an area without changing the other slots for instance. These are mainly user interface changes, as they would simply correspond to altering the initial constraints.

Also, smarter backtracking could speed up convergence by handling the cases of dead end a bit differently: rather than restarting from the initial configuration, i.e., cancelling all observations, one could cancel only the last n ones, the challenge being to determine n. A value of 1 would mean to traverse the graph of solution depth-first and could take a significant time to jump away from a bad branch of search.

A deeper change to the tiling algorithm could be to solve for interfaces first, and then generating the list of tiles with all the configuration occurring in the result. With a procedure to limit the number of such configurations during solving, this corresponds to merging the tiling engine and our tile suggestion mechanism into a single algorithm, ensuring that there is always a solution but at the cost of less control given to the user over the tile set.

IV.3 Parametric tile content

IV.3.1 Introduction

We saw that tiling systems are a powerful framework for content generation, thanks to their ability to produce large aperiodic extents from a small dictionary of elements. And their constraint-based nature makes them easily user-directable. However, tile-based content often suffers from visual redundancy and rigidity, even when applied on a non-regular grid. To address this, we augment the discrete tiles of the input dictionary with scalar parameters driving variations of their content, like for instance the height marked as variable in Figure I.6, making them *parametric tiles*.

The key challenge induced by this change is that tile assignments and hyper-parameter values must be solved simultaneously. In some easy cases, the hyper-parameter valuation is simply a post-process, but as illustrated in Figure IV.13.b there exists more intricate scenarios. Here the range of allowed value for h,

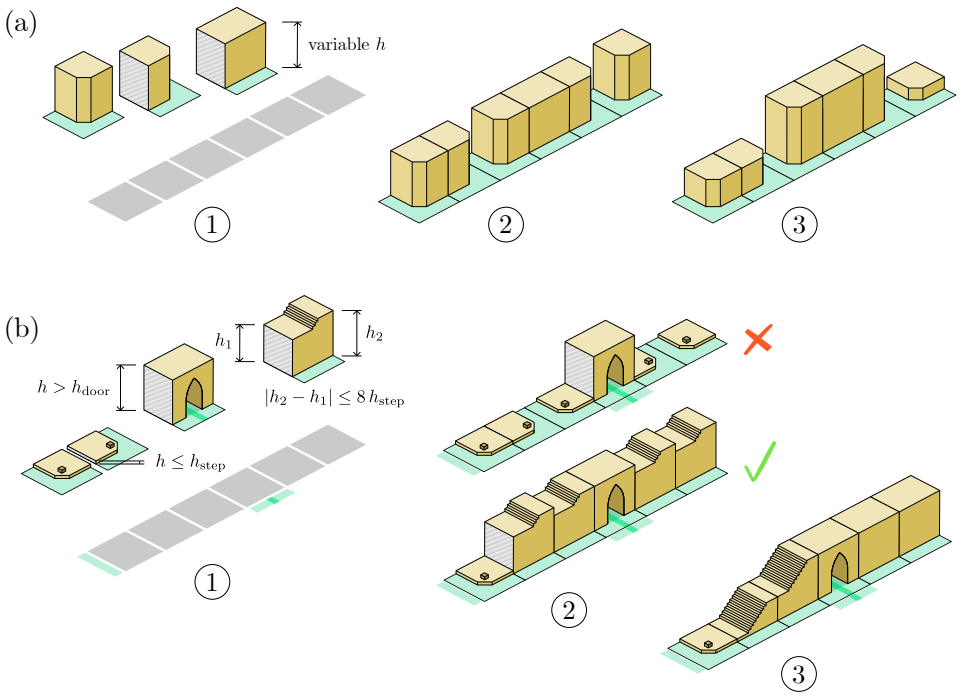


Figure IV.13: *Like any tiling problem, we start from a set of tiles and a graph of slots* ① *. In simple a scenario like (a), assigning a value to the hyper-parameter h of parametric tiles can be thought as a post-process* ③ *independent from the tiling itself* ② *. However, in presence of more advanced constraints on h, like in (b), these must be taken into account during the tiling. Otherwise there is no way to detect in* ② *that the top assignment cannot lead to a valid solution whereas the bottom one can, as shown in* ③ *.*

combined with the constraint that adjacent tile interfaces must have the same height, make the discrete tiling engine unable to distinguish between a valid solution and one that will lead to a dead-end when assigning values to h. We thus designed a *parameter-aware tiling algorithm*, which handles tile neighboring constraints that are affecting the range of validity of scalar variations.

Contributions The main challenges we address are (IV.3.2.1) the expression of constraint propagation equations using continuous tile sets, in a way that can be implemented in practice, namely without infinite unions, then (IV.3.2.2) the definition of a compact representation of infinite tile superposition and (IV.3.2.3) a procedure to sample this representation, for the *observe* step of the WFC algorithm.

IV.3.2 Method

We adapt the tiling engine presented in Section IV.2.2.3, based on the algorithm of WFC (Merrell, 2007; Gumin, 2016). We must however change the formulation

of the tiling problem that was given in Section IV.1.2.1, because the set T of tiles, which was only discrete, now contains both discrete and continuous variations of the tiles. And the set I of interfaces also undergoes this shift. In general, if $T_\diamond$ is the discrete set of tile types and each type $t \in T_\diamond$ is a parametric shape of hyper-space parameter Π_t, then the continuous tile set is:

$$T = \bigcup_{t \in T_\diamond} \{t\} \times \Pi_t \tag{IV.1}$$

We note $t(\boldsymbol{\pi})$ an element $(t, \pi_1, \dots, \pi_K)$ of T. In the example of Figure IV.13.b, we have tile types $T_\diamond = \{BeginWall, EndWall, Door, Stairs\}$ and for instance $\Pi_{Door} = [h_{\text{door}}, +\infty[$.

Interface-based parameterization Similarly, and more importantly, the set of interfaces becomes $I = \bigcup_{i \in I_\diamond} \{i\} \times \Pi_i$ where $i \in I_\diamond$ is a parametric interface type and Π_i is its hyper-space parameter. These *interface* hyper-parameters are what matters to the tiling engine. On the contrary, the *tile* hyper-parameters that do not affect any interface can be handled in a post-process, so we ignore them and, without loss of generality, we consider that

$$\Pi_t \subset \Pi_{i_N} \times \Pi_{i_S} \times \dots$$

where $i_d = L_\diamond(t, d)$ is the interface type that labels tile type t in direction d.

In the example of Figure IV.13.b, interfaces types are *Wall*, *Ground* and *Road*, with $\Pi_{Wall} = \mathbb{R}^+$. The tile type *Stairs* is parameterized by $(h_N, h_S) \in \Pi_{Wall} \times \Pi_{Wall}$ because both its north and south interfaces are *Wall*. More generally, we note tiles $t(\pi_N, \pi_S, \dots) = t(\boldsymbol{\pi})$.

> **NB** What we assume in the end is that the labeling function $L : T \times D \to I$ is separable into its discrete part $L_\diamond : T_\diamond \times D \to I_\diamond$ and its continuous part $((\pi_N, \pi_S, \dots), d) \mapsto \pi_d$, such that $i(\dots) = L(t(\dots), d)$ only if $i = L_\diamond(t, d)$. In other terms, the interface *type* i of a tile type t does not depend the on value of the hyper-parameters.

IV.3.2.1 Constraint propagation

The main step that gets affected by the continuity of the tile set is the constraint propagation, a.k.a. the *collapse* step in WFC's wording.

We have two types of constraints. First, discrete neighboring constraints – stating that interfaces types which are in contact must match – are the same as in the original discrete tiling problem. We write $i \equiv i'$ to mean that interfaces i and i' are

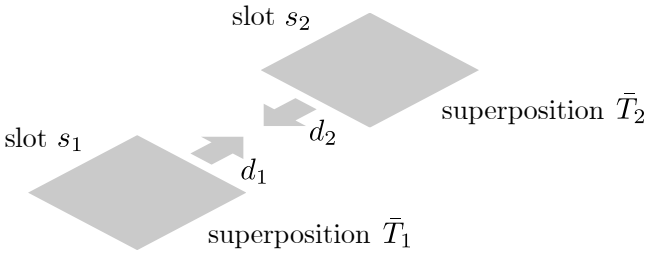


Figure IV.14: *The slot s_2 is in direction d_1 with respect to the slot s_1, and inversely the slot s_1 is in direction d_2 with respect to slot s_2. We note $d_1 \leftrightarrow d_2$ the edge going from s_1 to s_2.*

allowed to be in contact, to cover both cases where we need $i = i'$ and cases like in Section IV.2.2.3 where it is $i = \underset{\leftrightarrow}{i'}$.

Secondly, there are hyper-parameter constraints. This is simply expressed as an equality between the hyper-parameters of i and these of i', and thanks to our interface-based parameterization, it directly translates into constraints on the hyper-parameters of tiles.

In order to propagate both types of constraints, we need to adapt *ResolveConflicts*(s_1, d_1, s_2, d_2) in Algorithm 5. This call updates the superposed assignment $\bar{T}_2 \subset T$ of slot s_2 by keeping only the tiles that are allowed to be next to the tiles of $\bar{T}_1$ in the direction $d_1 \leftrightarrow d_2$ (Figure IV.14):

$$\bar{T}_2 \leftarrow \bar{T}_2 \cap \mathit{Allowed}(\bar{T}_1, d_1, d_2)$$

In a discrete tiling problem, we usually pre-compute a table $R[i, d] = \{t \mid L(t, d) \equiv i\}$, and then evaluate $\mathit{Allowed}(\bar{T}_1, d_1, d_2)$ as:

$$\mathit{Allowed}(\bar{T}_1, d_1, d_2) = \bigcup_{t \in \bar{T}_1} R[L(t, d_1), d_2]$$

In the parametric tiling problem, it is no longer possible to store the table R, nor it is possible to iterate through $\bar{T}_1$. Using i_k as a shorthand for $L_\diamond(t_k, d_k)$, we write:

$$\mathit{Allowed}(\bar{T}_1, d_1, d_2) = \bigcup_{t_1(\boldsymbol{\pi}_1) \in \bar{T}_1} \left\{ t_2(\boldsymbol{\pi}_2) \mid i_2 \equiv i_1 \text{ and } \boldsymbol{\pi}_{2d_2} = \boldsymbol{\pi}_{1d_1} \right\} \tag{IV.2}$$

Any formulation of this set suffers from the presence of an infinity of tiles, but we express it in a way that better suits the representation of tile superposition $\bar{T}$ that we introduce in Section IV.3.2.2:

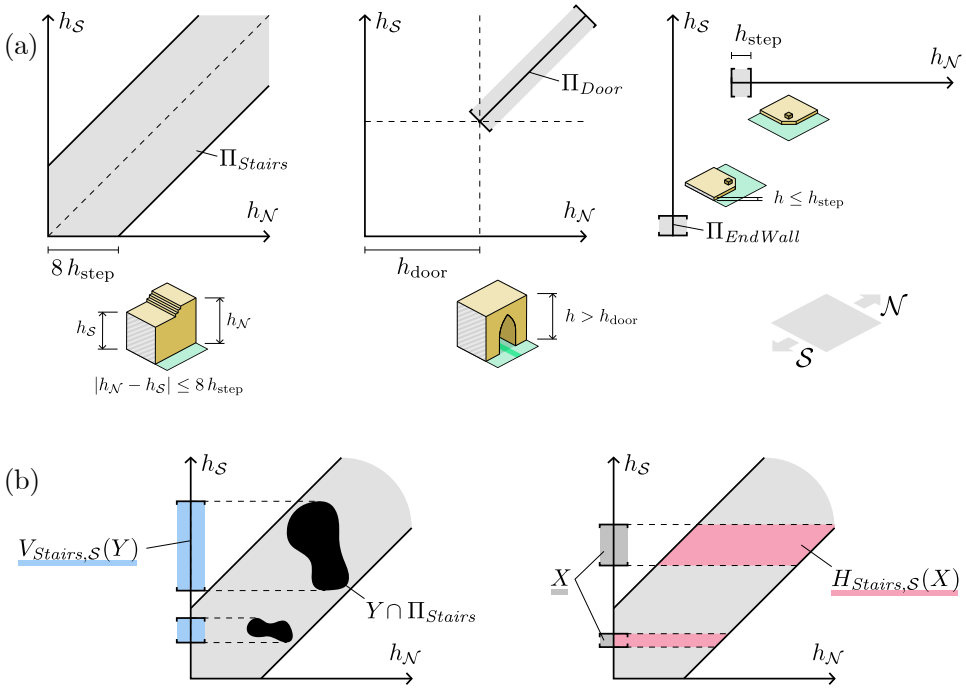


Figure IV.15: *(a) Hyper-parameter spaces for tiles of Figure IV.13.b expressed using the interface hyper-parameter h. (b) Illustration of tile set projection V and its pseudo-inverse H.*

$$Allowed(\bar{T}_1, d_1, d_2) = \bigcup_{t_2 \in T_\diamond} H_{t_2,d_2}\Big(\bigcup_{\substack{t_1 \in T_\diamond \\ \text{st. } i_1 \equiv i_2}} V_{t_1,d_1}(\bar{T}_1) \Big) \tag{IV.3}$$

We note $H_{t,d}(X) = \{t(\boldsymbol{\pi}) \mid \pi_d \in X\}$ the set of variations of a tile type t whose interface in direction d has its hyper-parameters in X. And we note $V_{t,d}(Y) = \{\pi_d \mid t(\ldots, \pi_d, \ldots) \in Y\}$ the set of interface hyper-parameters exposed in the direction d by the tiles of type t in the superposition $Y \in T$. In other terms, V is the projection of $\Pi_t \cap Y$[1] onto Π_i, with $i = L_\diamond(t, d)$, and H is its inverse:

$$V_{t,d}\left(H_{t,d}(X)\right) = X$$

These functions are illustrated in Figure IV.15 and the derivation from Equation IV.2 to IV.3 is detailed in Appendix C. The strength of the formulation of Equation IV.3 is that the unions can be implemented as finite loops. The next section shows how we encode the infinite sets returned by H and V.

[1]This is a shorthand for $(\{t\} \times \Pi_t) \cap Y$, if we refer to the definition of T of Equation IV.1

IV.3.2.2 Representation of a tile superposition

We represent a superposition $\bar{T}$ of tiles as a series of axis aligned bounding boxes. For each tile type $t \in T_\diamond$, we store a single bounding box $B_t = [a_1, b_1] \times [a_2, b_2] \times \ldots$:

$$\bar{T} = \bigcup_{t \in T_\diamond} \{t\} \times B_t$$

Our representation contains false positives, namely tiles that are not in the actual superposition but within the bounding box anyways, but it does not have any false negative. This means that the algorithm will not miss branches of solutions, but since we reduce the possibility space more slowly than with a perfect representation the algorithm may need more time to finish.

V and H are implemented for bounding boxes only. $V_{t,d}(B_t)$ is simply the restriction of B_t to the axes of direction d. Functions $H_{t,d}$ are what actually contains the rules relating the hyper-parameters of a tile, and are considered as user input. For instance, for tile *Stairs* in Figure IV.15:

$$H_{Stairs,\mathcal{S}}([a, b]) = [a - 8\, h_{\text{step}}, b + 8\, h_{\text{step}}] \times [a, b]$$

For simple constraint equations like in this case, H can be derived using symbolic calculus:

```
>> from sympy import *
>> hs, hn, hstep = symbols('h_S, h_N, h_step', positive=True)
>> constraint = Abs(hs - hn) < 8 * hstep
>> solveset(constraint, hn, domain=S.Reals)
Interval.open(h_S - 8*h_step, h_S + 8*h_step)
```

Listing IV.1: *The python package `sympy` can be used to derive the expression H from a simple inequality.*

IV.3.2.3 Sampling tile superposition

During the *observe* step of the WFC algorithm, we need to sample a single tile $t(\boldsymbol{\pi})$ from a superposition $\bar{T}$, and this tile must not be a false positive. We do this by sampling hyper-parameters incrementally, refining the bounding box after each one to remove obvious false negatives. And for each hyper-parameter, we use either one end of the bounding box or the other, no in-between. If the density of false positives is low enough, another strategy is dart throwing, i.e., sampling repeatedly until the parameter $\boldsymbol{\pi}$ is valid, but none of our examples use this sampling scheme.

IV.3.3 Results

Switching the representation of superposition in assignments A of Algorithm 5 to the bounding boxes described in Section IV.3.2.2, and adapting the collapse and

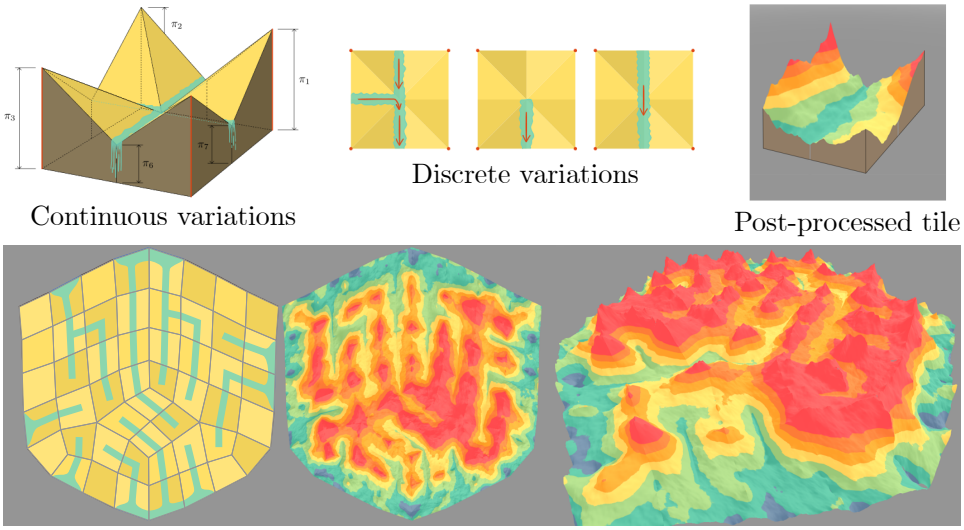


Figure IV.16: *Watershed network generated using our parametric tile engine. Our parametric tiling engine ensures that the water flows consistently and assigns altitudes at each interface.*

observe steps according to respectively Section IV.3.2.1 and IV.3.2.3, we are able to run the *Parametric WFC* algorithm on continuous tile sets.

IV.3.3.1 Watershed generation

In the example of Figures IV.16 and IV.17, we generate watershed using tiles parameterized by the altitude of their corners and the middle of the edges. Each tile is constrained such that the water flows in the expected direction, and the discrete interface matching $\equiv$ ensures that water is flowing in the same way across boundaries. In this example, corner altitudes are defined in a post-process, but waterway altitudes are needed for the tiling. In order to generate an island, we constrain the altitude at boundaries to 0. Procedural fractal variations are finally added to bring more diversity. Compared to other landscape generation algorithms, this approach ensures the creation of meaningful watersheds and consistent large scale features like valleys.

Other approaches, such as the work of Génevaux et al. (2013), use hydrology as a driver for terrain generation. We do not intend to compete with these but rather use their scenario for illustration of our more generic algorithm. Large scale features in terrain generation can also be generated from tectonic activity (E. Michel et al., 2015; Cordonnier et al., 2018).

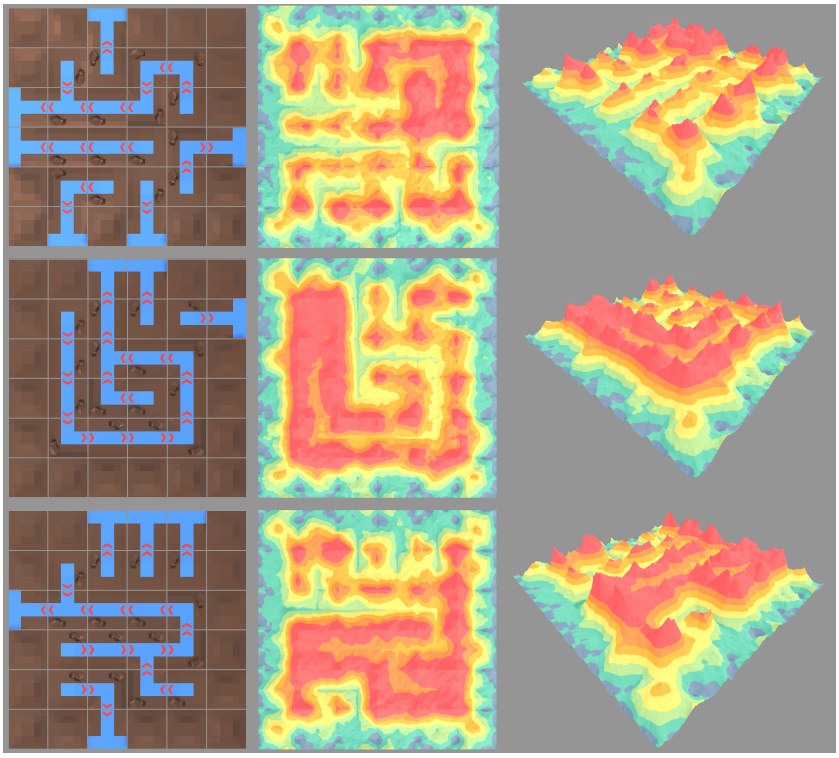


Figure IV.17: *Multiple runs of our watershed generation based on parametric tiles.*

IV.3.3.2 Discussion

Our method shows that the core mechanism of the WFC algorithm generalizes to continuous tile sets, namely tile sets whose tiles are parametric shapes. This resonates with our chapter on imperative programming (Chapter III), since such tiles are typically described as DAGs.

In order to handle continuous tile sets, we needed a simplification hypothesis for representing superposition of tiles. Nevertheless, Section IV.3.2.1 makes no extra assumption, so the equations based on functions H and V come without loss of generality and might be reused together with a different representation of tile superposition.

The choice of a single bounding box, i.e., a single range per hyper-parameter, is a trade-off between expressivity and space requirement. Our continuous representation takes obviously more space than in discrete WFC, where it can be represented as a compact bitfield (using one bit per tile), but way less than a more accurate representation. This model becomes problematic when the parameter set of a tile is made of multiple connected components; we advise in such a case to consider each component as a different tile type (as long as there is a finite set of connected components).

V

Visual feedback of shape programs during authoring

V.1 Introduction

V.1.1 A two-way integration of rendering and generation

As introduced in the Chapter I, our interest for program-based representations of shapes is grounded in their ability to support the creation process. We focus on shape programs that evaluate in interactive time, but sometimes the technical limitation to interactivity comes from the real-time rendering of its output.

It does not take a long shape program to generate heavy content, so the limitations of real-time rendering are hit faster when designing shape through its program-based representation than when building it manually. Fortunately, we can use the program itself as a mean to detect and optimize geometrical redundancies in the shapes.

We presented in Chapter II multiple examples of procedural modeling systems that include considerations about rendering, for shape grammars representing building facades (Haegler et al., 2010) or whole cities (Steinberger et al., 2014), for fiber-level garment modeling (K. Wu & Yuksel, 2017), or for CSG-based modeling (Goldfeather et al., 1986; Kirsch & Döllner, 2004; Zanni et al., 2018).

In this chapter, we study two ways to blend the shape program with its real-time rendering pipeline. In Section V.2, we delay the evaluation of some parts of the shape program, namely the tile instancing and deformation, to benefit from hardware acceleration and enable real-time visualization of the complex mesostructures generated in Section IV.2. In Section V.3, we have the shape program output multiple first-order representations, which we combine information about the

viewpoint to provide real-time feedback at multiple level of details (LoD) for large self-repeating aggregates of quasi-spherical elements.

V.1.2 Related works and background

Level-of-Detail LoD methods intend to generate simplified versions of a complex object that are visually equivalent at a given distance while computationally lighter. Surfacic mesh simplification methods, either based on repeated contractions of edges guided by some cost function (Hoppe et al., 1993; Hoppe, 1996; Garland & Heckbert, 1997) or by spatial clustering (Rossignac & Borrel, 1993), have become standard LoD methods and can even be applied to very large meshes (Lindstrom, 2000). However, as pointed out by Cook et al. (2007), such methods fail when the geometry is an aggregation of already simple elements, which vanish if simplified further.

Volumetric models also have their LoD mechanisms. On voxel-based models, the SGGX distribution (Heitz et al., 2015) and follow ups (Zhao, Wu, et al., 2016; Loubet & Neyret, 2018) have enabled techniques for downsampling a volume without altering its visual appearance. Hierarchical structures can be used to organize data in a tree whose traversal is dynamically adapted to the view point, either with voxels (Crassin et al., 2009; Kämpe et al., 2013) or with points (Rusinkiewicz & Levoy, 2000; Gobbetti & Marton, 2005). Most of these techniques assume static geometry though, which is not compatible with the viewport of an authoring tool.

All these LoD methods are designed for a single class of model. Sequential point trees (Dachsbacher et al., 2003) are an interesting evolution of QSplat (Rusinkiewicz & Levoy, 2000) using an hybrid model, but it makes the same assumption that the point cloud is static. The inter-model transition was recently successfully addressed in the surface-to-volumetric context by Loubet and Neyret (2017). Their setting is more general than ours but designed for off-line rendering and not tacking advantage – because not assuming – of self-similarity.

In Section V.3, we focus on dynamic element positions, depending on input hyper-parameters. This prevents us from using techniques that precompute clusters of geometry to merge, like *Occluder Fusion* (Wonka et al., 2000) or *CellVIEW* (Le Muzic et al., 2015). The latter is a case of molecular visualization, which generally involves LoD of dense aggregates of spheres that motivated dedicated research, as surveyed by Miao et al. (2019). Although such visualization techniques deal with static perfect spheres, usually uniformly colored, setting them aside from many issues we intend to tackle here, they need to handle very large amounts of atoms for which they develop inspiring advanced drawing strategies.

Impostors One of the most extreme simplification consists in using *billboards*. A billboard, or planar impostor, is made of one single plane, and its whole appearance is encapsulated in (the maps of) its material, with its perceived shape

being expressed by its silhouette, reproduced using transparency. The extreme simplicity of a billboard's geometry allows to invest more resources in shading, with its associated material containing information about the normal field of the original geometry, and even the depth component leveraged by *relief mapping* techniques (Policarpo et al., 2005).

The limits of a single billboard are quickly reached, usually because of the limited range of directions for which it is valid, but they are at the root of many lightweight approximation models in computer graphics. Aggregated billboards are often called *multi-view impostors* since they address the view dependency of planar billboards. Maciel and Shirley (1995) build a LoD hierarchy in which billboards are precomputed for some key directions. Billboard clouds (Décoret et al., 2002) extract several billboards using a Hough transform to approximate a high definition mesh model. A typical use case of billboards, and hence multi-view impostors, is tree rendering, like Meyer and Neyret (2000) and more recently Bruneton and Neyret (2012) whose method is related to our Section V.3, though their model remains surfacic at all scales and they use impostor for the different goal of accounting for foliage's semi-transparency.

Todt et al. (2007) provide a good overview of the possible parametrizations of a spherical impostor, however they focus on a different use case where a single complex model is rendered, leading them to different design choices. In particular, their selection of precomputed directions, and advanced compression, projections and intersection refinement schemes, while saving memory, quickly becomes too prohibitive to apply for each grain in our scenario. Some of these limitations are addressed by Brucks (Brucks, 2018) who, similarly to our approach, also make the impostors dynamically relightable by storing maps that represent the attribute field (like the G-Buffer) rather than a static grain light field. However, Brucks renders order of magnitude less impostors than in our use case, so they can still afford storing depth maps and computing relief mapping. Since they use it for trees, they also deal with significantly smaller inter-impostor occlusion.

Filtering Filtering attribute-encoding images, as mandatory with mipmapping, is not trivial for attribute with non-linear response, such as normal and roughness maps. This issue has been addressed by Tan et al. (2008), *LEAN mapping* (Olano & Baker, 2010) and then *LEADR mapping* (Dupuy et al., 2013), which are compatible with our method. More recent works even try to adapt the concept of mip-maps to the BSDF itself rather than to its attribute maps (C. Xu et al., 2017).

V.2 Tile-based Mesostructure Rendering

Section IV.2 presented a tile-based method for authoring mesostructure geometry. Our approach enables designers to produce an heavy amount of geometrical content, easily reaching hundreds of millions of triangles when represented as a

Figure V.1: *Captures from the real-time viewport of our mesostructure authoring tool.*

mesh. Since we intend to provide an interactive authoring system, we need a way to render this geometry in real time, similarly to what other surface amplification methods, like displacement mapping (Szirmay-Kalos & Umenhoffer, 2008) and subdivision surface (Brainerd et al., 2016), do.

When facing a similar situation in Section V.3, we will choose to make the shape program output a different representation of the geometry, namely impostors or point clouds, but the topological complexity of mesostructures makes the use of simplified meshes, impostors and other LoD techniques unfit for our current problem. Instead, we use a full geometry but complete its generation on-the-fly within the render pipeline.

V.2.1 Method

V.2.1.1 Render Pipeline

We start by sampling each 2D cross-section of each interface with a list of points using *Clipper* (2014). These cross-sections are stored as CSG trees modeling a 2D space occupancy function during editing so that we can change the discretization to a user defined resolution target dynamically. The resulting points sets are then stored in 1D texture maps using a *repeat* wrap mode.

Second, for each sweep surface in the tile set – not for each instance – we allocate a GPU vertex array object (VAO) modeling a regular grid mesh. The horizontal resolution of this grid is the maximum of the size of the start and end cross-

section textures. The vertical resolution is a user defined parameter driving the smoothness of the sweep objects. We use a compute shader to assign x and y coordinates to each point by interpolating from the start to the end section, taking care of reversing the coordinate at which cross-section textures are sampled from u to $1 - u$ when an interface is flagged as *flipped*. This creates base sweeps that will later be deformed per-instance to conform to their target trajectory.

Third, the shell space is represented in GPU memory as a buffer (SSBO) storing, for each macrosurface quad, the eight corners of its shell hexahedron.

Fourth, we allocate four SSBOs to hold the control points of the Bézier curves, containing one vector per instance of a sweep. A compute shader uses the shell space SSBO and the slot assignments to fill these control point buffers.

Finally, one draw call is issued for each type of sweep surface, and hardware-instanced as many times as there are uses of its parent tile type in the slot assignment. We deform the VAOs at the vertex shader stage to follow the Bézier trajectory. Any shading method can be used on the rasterized fragments.

V.2.1.2 Caching

Our rendering pipeline caches the result of the previous frame as much as possible:

- A cross-section texture is modified only if the corresponding shape has been edited by the user.
- A sweep VAO is recomputed only if the sweep is new or if one of its cross-sections has been modified.
- The shell space is uploaded only when a new macrosurface is used (or if it is deformed on CPU).
- The positions of the control points are recomputed only when the slot assignments change, or when the shell space evolves (the thickness and offset user parameters).

V.2.2 Results

V.2.2.1 Performance

The performance of our C++/OpenGL prototype are reported in Table V.1, measured with an Intel Core i5 CPU, with 16 GB of RAM and an NVidia GeForce Titan RTX. As an element of comparison to show the compactness of our representation, the example (1c) occupies 1.75 GB when exported as a binary PLY file. Additional figures are reported in Appendix D.

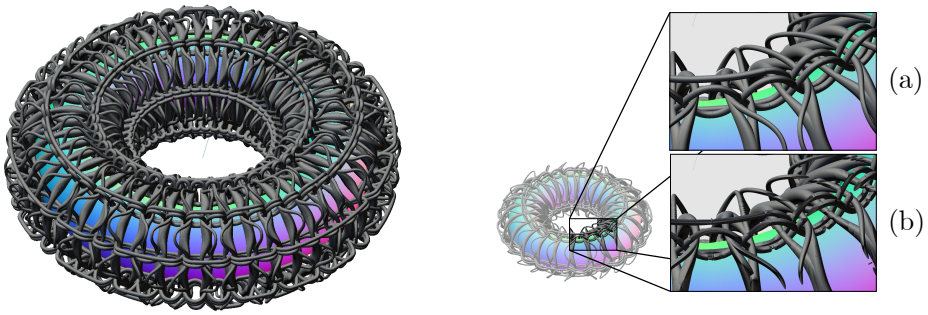


Figure V.2: *Our mesostructure rendered as a signed distance field, using sphere tracing. Inset (a) and (b) show the visual artefacts that arise from this representation, (b) being voluntarily degraded for illustrative purpose.*

Table V.1: *For each example of Figure IV.10, the amount of GPU memory required to store our model, the number of drawn triangles, the corresponding render time and the time need by the tiling engine.*

Example	Memory	Triangles	Render	Tiling
(1a)	5.66 MB	57.9 M	6.2 ms	1132 ms
(1b)	3.06 MB	44.7 M	6.0 ms	3212 ms
(1c)	6.30 MB	70.0 M	7.9 ms	708 ms
(2a)	2.96 MB	24.3 M	5.1 ms	87 ms
(2b)	1.83 MB	18.8 M	5.1 ms	135 ms
(2c)	4.18 MB	29.8 M	5.8 ms	279 ms
(3a)	1.76 MB	6.42 M	3.6 ms	17.8 ms
(3b)	1.33 MB	7.19 M	4.0 ms	43.3 ms
(3c)	3.49 MB	6.18 M	3.1 ms	39.1 ms

V.2.2.2 Surface representation

The procedural nature of our core representation enables us to synthesize various representations of the surface of the end mesostructure. We have focused on traditional meshes (Figure V.1), but this choice creates self-intersection when multiple sweeps start from the same profile (Y joints). An alternative choice consists in creating implicit surfaces, which handle sweep volumes rather well (Schmidt & Wyvill, 2005). Early tests (Figure V.2) show that a promising approach consists in an hybrid representation where the macrosurface's shell is rasterized and then tile's content is drawn using sphere tracing. Although preventing self-intersections and producing nice blends, our tests show slower rendering time compared to the mesh representation.

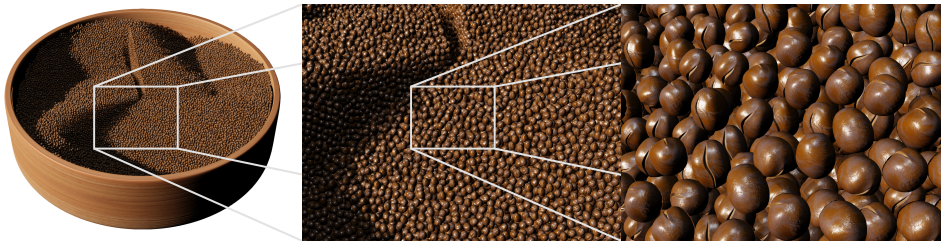


Figure V.3: *Our level-of-detail method exploits quasi-spherical impostors to render, in real time, fully dynamic stackings made of millions of similar objects, with variable materials and orientations, while seamlessly integrating into deferred shading.*

V.2.3 Discussion

V.2.3.1 Properties

Our entire mesostructure synthesis runs almost fully on the GPU, leaving the CPU largely available to the tiling engine and achieving hundreds of millions mesostructure polycount at real-time framerate. The parametric nature of our representation makes it possible to dynamically adapt the resolution of the mesh it produces, allowing manipulation on lower end devices, and opening a potential for LoD mechanisms.

V.2.3.2 Future work

Delaying the evaluation of some stages of the shape program is a very efficient approach to enable real-time feedback in multiple scenarios. It is however hard to automatically figure out which part of the geometry generation may be off-loaded to GPU compute shaders and how. The most efficient solution often remains to hand tune the integration of shape program and real-time rendering, like we are doing here. Another common scenario is the case of a DAG whose last operation is a subdivision surface: it is often evaluated in an hardware-accelerated tessellation stage. And when a DAG ends with an instancing, this is usually hardware accelerated, or tuned even further as we see in the next section.

We could go further by handling more common cases and matching patterns in the DAG to port them to render-time on-the-fly evaluation, or chose to implement mesh processing effects using dedicated languages able to abstract this such as Halide (Ragan-Kelley et al., 2012).

V.3 Multiscale Rendering of Dense Dynamic Stackings

V.3.1 Introduction

Fruits in a market-place, coffee beans in a roaster or bolts at the hardware store are typical examples of stackings found in 3D scenes and generated by a program

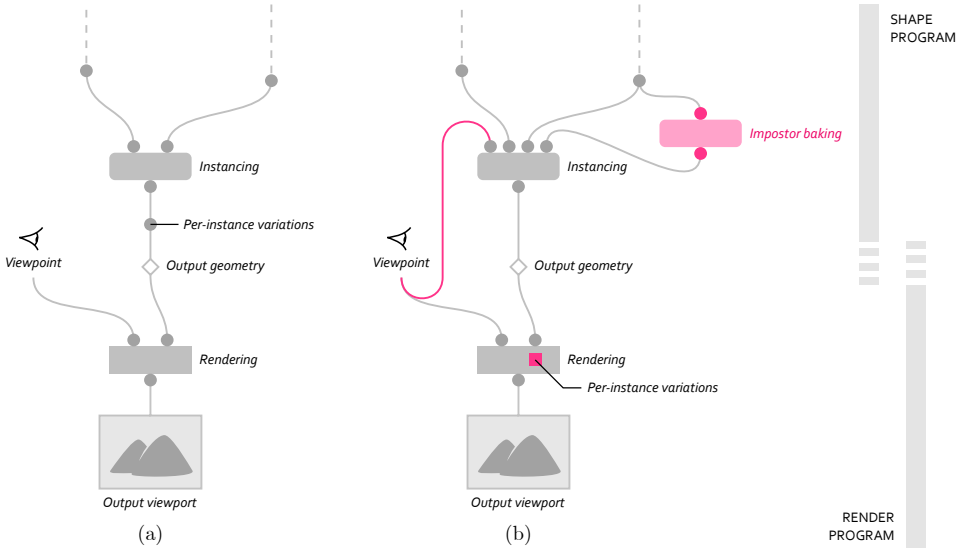


Figure V.4: *(a) When a shape programs ends with the dense instancing of some quasi-spherical objects, (b) we feed viewpoint information to the instancer and have the shape program compute a different representation of the instances, namely impostors. Like in Section V.2, the evaluation of per-instance variations (size, orientation, material) is delayed to the render program.*

that ends with instancing (Figure V.4). They challenge LoD mechanisms to achieve both high speed rendering and detail preservation. At each extremity of the LoD chain, existing methods are well covered by the literature. Closer views are better handled using a mesh-based model that can be progressively simplified (Hoppe, 1996) while further views leverage point-based rendering (Gross & Pfister, 2007). But none of these models fits well the transition phase, when stacked elements – which we call *grains* in the reminder of this section – cover tens to hundreds of pixels. Under this regime, mesh-based simplification makes whole elements vanish when pushed too far, while point-based rendering lacks high frequency details that should still be clearly visible.

The self-similarity of the stacking naturally leads to per-grain *impostors* for this transition scale. When the number of visible grains is very large compared to the number of possible view angles, it becomes worth precomputing a few views and then, at runtime, picking for each grain the closest one. There are two ways to describe an impostor, either as a projection of the geometry of a grain or as a rich splat. The former relates to mesh-based models while the latter relates it to point-based graphics, which suggests this is a good candidate as a transition model.

Impostors come with their own limitations, e.g., memory consumption, hardware support or overdraw, that we propose to overcome making three assumptions that

stem from the typical properties of stacked grains:

- **quasi-spherical shape**: The grain's surface is bounded between two co-centered spheres, namely an inner sphere of radius r and an outer sphere of radius R; the closer these radii, the more efficient our approach.
- **moderate shape diversity**: All grains share the same (or only a few) silhouette, which prevents memory consumption and improves caching.
- **density**: occlusion culling becomes more impactful as density increases, even if approximate, as long as grains don't intersect each others.

Fortunately, the loss of generality induced by these hypotheses is in practice largely mitigated as noticed by Moon et al. (2007) and Meng et al. (2015). The diversity of shape can be increased by arbitrary scale/rotation of each grain together with procedural variations of its material attribute maps. Our approach is oblivious to the grain material model: in practice, we exemplify our method on standard microfacet models rendered using deferred shading. Moreover, relaxing the quasi-spherical or density hypothesis only leads to progressively degraded performance, but not to any gap in visual appearance. Based on these assumptions, we make the following contributions:

- a real time splitting process of the input grain set into per-scale/drawing model buffers, leveraging an analysis of when to split (Section V.3.4.1) and how to do it (Section V.3.4.2),
- a sampling scheme for the impostors suited to our quasi-spherical proxy (Section V.3.3.3) and improving their visual appearance w.r.t. ground truth,
- a novel occlusion culling mechanism tailored for dense stackings of quasi-spherical objects (Section V.3.5.1), that helps alleviating rendering prior to determining the exact grain shape,
- an efficient rendering pipeline for a cloud of many impostors based on early-Z rejection (Section V.3.5.2) – a hardware mechanism not natively suitable for impostors, whose actual shape remains unknown up to the sampling of their maps.

As a result, our approach is versatile enough to model various use cases and scales well up to extreme amounts of grains such as in sand rendering.

V.3.2 Pipeline overview

Figure V.5 describes the sequence of draw events involved in rendering our aggregate of grains. The splitting step ① orchestrates rendering by routing each grain toward one model or another depending on its location. It discards some of them based on an occlusion map which is also further reused at step ③ to speed up drawing. Additionally, it early rejects grains out of the view frustum.

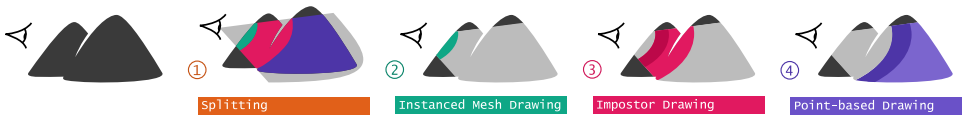


Figure V.5: *Anatomy of a dense stacking rendering sequence. A first step splits the stacking into several element arrays used to feed subsequent draw calls (Section V.3.4.2), also applying culling to early discard hidden points (Section V.3.5.1). Impostors are drawn in two passes (Section V.3.5.2). Point-based drawing typically discard all points beyond a limit distance. Our contributions concern steps ① and ③ .*

Models for close ② , mid ③ and far ④ grains are then rendered individually using the element buffers resulting from the splitting. We recall that, as a general rule of thumb, modern hardware-accelerated rasterization pipelines require closer elements to be rendered first to limit unnecessary fragment processing. Steps ② and ④ are the two models that we intend to bridge, respectively mesh-based and point-based, so we focus on the splitting process ① , which involves *when* (Section V.3.4.1) and *how* (Section V.3.4.2) to split the input point cloud, as well as on the impostor rendering ③ .

V.3.3 Impostors for dense stackings

Our mid-scale representation of the stacking is a cloud of impostors. The concept of spherical impostor is not new per se, but it can come with many flavors so we discuss which one is the best suited for real-time rendering of dense aggregates.

V.3.3.1 General rendering pipeline

We base our work on a spherical impostor model made of co-centered planar impostors facing different directions. Prior to rendering, the impostors are precomputed and then at render time, the only planes to be sampled are those whose normal vector is close enough to the view direction (Figure V.6).

Precomputation The impostor depends on the set of N view directions $(\omega_i)_{i=1...N}$ for which the grain's response is precomputed. For each view index i, a $p \times p$ sprite of the grain is rendered from a view point in direction ω_i, storing for each pixel the material attributes (albedo, roughness, normal, etc.) in an atlas of maps $(I_i)_{i=1...N}$ where I_i is the response at different positions of the sphere in direction ω_i.

Runtime In order to reduce as much as possible the geometric footprint of the impostors, we use simple sprites, e.g., OpenGL's GL_POINTS, as our drawing primitive. The sprite size is computed in the vertex shader to ensure that it covers the whole outer sphere of the grain. When drawing the impostor, we fetch the object's appearance attributes at a given point in a given direction. The main steps

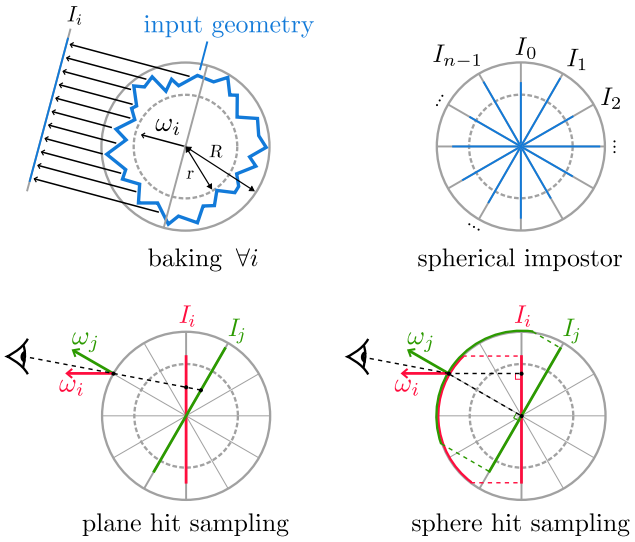


Figure V.6: *A spherical impostor is a set of concentric planar impostors I_i precomputed for different directions ω_i, as well as an inner radius r and outer radius R. At render time, we use only the most relevant ones. They can be sampled as planes (bottom left) or as hemispheres (bottom right), or using our mixed sampling scheme (Section V.3.3.3).*

of impostor sampling are **(i)** to seek for the indices of the appropriate precomputed views (planar impostors) given the orientation and position of the grain in camera space, **(ii)** to sample the right texel from the impostor maps and **(iii)** to interpolate the responses of different planes. The interpolation weights ensure the visual continuity by progressively fading out the contribution of a plane when the view point changes. In the design of such a sampling, two choices can have a major impact: the *parametrization*, and the *definition*, i.e., the density of the sampling. The latter is discussed in Section V.3.4.1 when analyzing the bias introduced by the impostors.

V.3.3.2 Parametrization

In order to still benefit from hardware texture filtering (mipmaps), especially to reduce aliasing when grains become very small on screen, we use the parametrization that Todt et al. (2007) calls Sphere-Plane. When sampling the atlas of precomputed views, the view index represents a direction and the texel coordinate a position offset, not the other way around. In practice, this means that precomputed views are rendered using orthographic cameras.

There remains to decide on the directions to precompute. The list (ω_i) of such directions must verify several conditions:

- **coverage**: There must always be a billboard close enough to the viewing direction among the precomputed atlas.

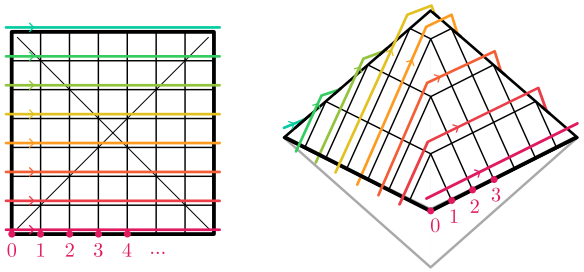


Figure V.7: *We precompute view directions using vertices of a subdivided octahedron (the L_1 sphere) since mapping them to integer indices is computationally efficient; here with $n = 8$ vertices by edge boils down to $N = 128$ views.*

- **compactness**: Each precomputed view has a video memory footprint which must be small especially if there are many different types of grain.
- **speed**: To sample a given viewing direction, we need to efficiently determine the index of the closest view in (ω_i).

Coverage and compactness suggest an as regular as possible sampling of the unit sphere such as the distribution of Fibonnacci points (Keinert et al., 2015) or a statically optimized mesh (Todt et al., 2007). But speed is crucial in our scenario so we opt for a distribution based on a subdivided octahedron (see Figure V.7) as in Brucks (2018). Not only finding the index i of the closest view in this distribution is done in $O(1)$ with a little constant, but moreover it is easy to get the four closest views with their coefficients, which is important for interpolation.

At runtime, we compute the sampled directions at the extent of a whole grain, and not once for each fragment, i.e., we assume that camera rays are almost parallel for each fragments covered by a grain. Though this is inexact in general, it does not introduce strong distortion for grains whose extent on screen is limited to 100 pixels – our use case – and provides a significant speed-up.

```
void DirectionToViewIndices(
    vec3 d, uint n, out uvec4 i, out vec2 alpha
) {
    d = d / dot(vec3(1,1,1), abs(d));
    vec2 uv = (vec2(1, -1) * d.y + d.x + 1) * (n - 1) / 2;
    uvec2 fuv = uvec2(floor(uv)) * uvec2(n, 1);
    uvec2 cuv = uvec2(ceil(uv)) * uvec2(n, 1);
    i.x = fuv.x + fuv.y;
    i.y = cuv.x + fuv.y;
    i.z = fuv.x + cuv.y;
    i.w = cuv.x + cuv.y;
    if (d.z > 0) i += n * n;
    alpha = fract(uv);
```

```
14 }
```

Listing V.1: *Return in i the indices of the four closest precomputed views to the sampling direction d, assuming that the number of precomputed views is $N = 2n^2$, and in α the coefficients for interpolating between respectively $(i_0, i_2) \leftrightarrow (i_1, i_3)$ and $(i_0, i_1) \leftrightarrow (i_2, i_3)$. Note that instead of using the total number of views N, our procedure handles the number n of subdivisions along the edge of the octahedron.*

In practice, one can refer to the Listing V.1 for a GLSL implementation. The input direction d is normalized using the L_1 norm $L_1(d) = |d_x| + |d_y| + |d_z|$ and then converted to integer indices.

> **NB** In the atlas of a rich impostor, an index stores multiple maps, for the multiple attributes of the G-buffer. But they all conceptually share the same alpha transparency. Special care must be taken to account for alpha premultiplication when computing the mipmaps.

V.3.3.3 Sampling quasi-spherical impostors

Once a precomputed map I_i has been selected for sampling, different strategies may be adopted to decide which texel to read. We propose a sampling scheme that improves the visual appearance of under-defined impostors while remaining lightweight.

The most common choice is to assume that the the view direction is perfectly aligned with precomputed direction ω_i and compute the offset using intersection of the camera ray with the precomputed view plane. We call this *planar sampling* (Figure V.6, bottom left) and note P the texel it selects. In practice the camera and precomputed directions are not always well aligned, because we can store only a limited number of views. This results in ghosting artifacts, which particularly impact sharp visual features (Figure V.15) and stems from the distance between the plane and the actual geometry of the grain.

A second strategy consists in computing the intersection of the camera ray with an spherical proxy and then project this point along the precomputed view direction onto the precomputed plane (Figure V.6, bottom right). This *spherical sampling* gives another texel S. When using the average of r and R as radius of the sphere proxy, this greatly reduces ghosting, but cuts out parts of the object.

Therefore, we introduce a *mixed sampling* for quasi-spherical proxies. More precisely, we combine P and S depending on the relative distance d of the grain center to the camera ray normalized by the sphere's radius:

$$M = \frac{d}{R}P + (1 - \frac{d}{R})S$$

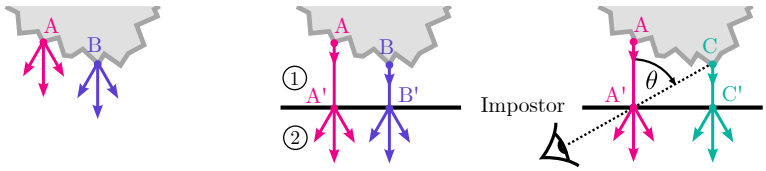


Figure V.8: *A simple planar impostor replaces original geometry (left) with a plane (middle). At baking time ① attributes are projected, then used at render time ② . This is valid up to a limit value of θ (right).*

This sampling succeeds at combining the benefits of both planar and spherical samplings, namely preserving silhouettes and sharp visual features. For a fixed memory budget and visual loss, this translates into more grains rendered as impostors and less as meshes, improving the overall performance.

V.3.4 Model discrimination

V.3.4.1 Impostors' validity range

We are not simply looking for a model that works at a given mid-scale, we also need to be able to smoothly transition from one model to another. In order to identify view conditions under which both mesh-based and impostor-based rending match, enabling us to substitute them, we must be able to quantify the range of validity of the impostor. This range of validity depends on the $p \times p$ amounts of spatial samples per view and the number $N = 2n^2$ of views precomputed for a mapping based on an octahedron with n subdivisions.

At the limit mesh-impostor distance L, the apparent grain diameter in pixels must match the size of the precomputed view, which gives us p proportional to R/L. The proportionality factor depends on the camera field of view and the screen resolution (see Appendix E for details). So from now on we assume that p is known and seek for N.

To do so, we need to know the maximum angle θ between a planar impostor's normal and the view direction for the impostor to return the right value. With the notations of Figure V.8, we look for the limit view angle beyond which A'C' exceeds the world space size $t = R/p$ of a texel, i.e., beyond which our model will sample the wrong texel. This constraint writes as follows:

$$A'C' \leq t \tag{V.1}$$

On another hand, the maximum distance between a point on the true surface of the grain and its projection onto the impostor is the outer radius R:

$$CC' \leq R \tag{V.2}$$

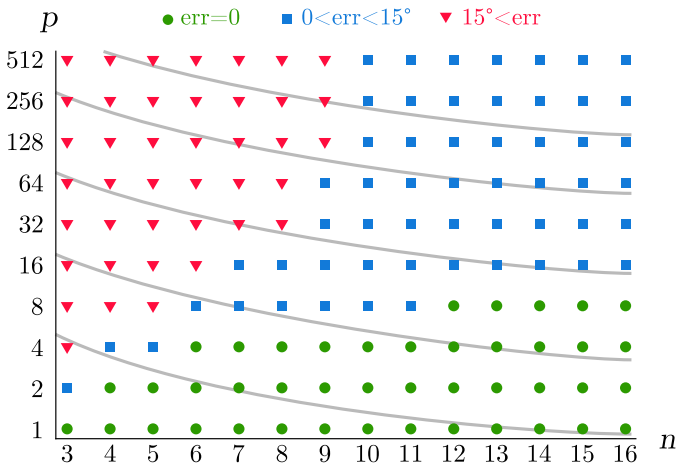


Figure V.9: *Mean angle error for different trade-offs of the two parameters n (subdivisions of the octahedron) and p (pixels per side) of a spherical impostor. Grey lines are iso-weight, i.e., two dots on the same line correspond to impostors occupying the same amount of video memory.*

This can be written using the angle θ between the impostor's normal and the view direction:

$$A'C' \leq R\sqrt{\frac{1}{\cos^2\theta} - 1} \tag{V.3}$$

To verify inequality (V.1), we can therefore look for:

$$R\sqrt{\frac{1}{\cos^2\theta} - 1} \leq R/p \implies |\theta| \leq \arccos\sqrt{\frac{1}{1+1/p^2}} \tag{V.4}$$

So each precomputed view is valid in a cone of angle $2\arccos\sqrt{\frac{1}{1+1/p^2}}$. This value has to be compared with the maximum angle between two neighbor points of the octahedron. Figure V.9 shows the evolution of this angle depending on the angular definition n and spatial definition p. More detailed tables can be found in Appendix E. In practice, we use less views than the theoretical threshold since our mixed sampling scheme (Section V.3.3.3) largely helps reducing artifacts for under-resolved impostors.

V.3.4.2 Dynamic grain splitting

Now that we know the range of validity of the impostor, we can dynamically discriminate the grain cloud into three subparts, namely the grains rendered using mesh instances, those rendered as impostors and the further ones rendered as points. We assume that all models are compatible with indexed rendering, which means that rather than when rendering K points, an *element buffer* of K indices can be provided to tell which grains to render.

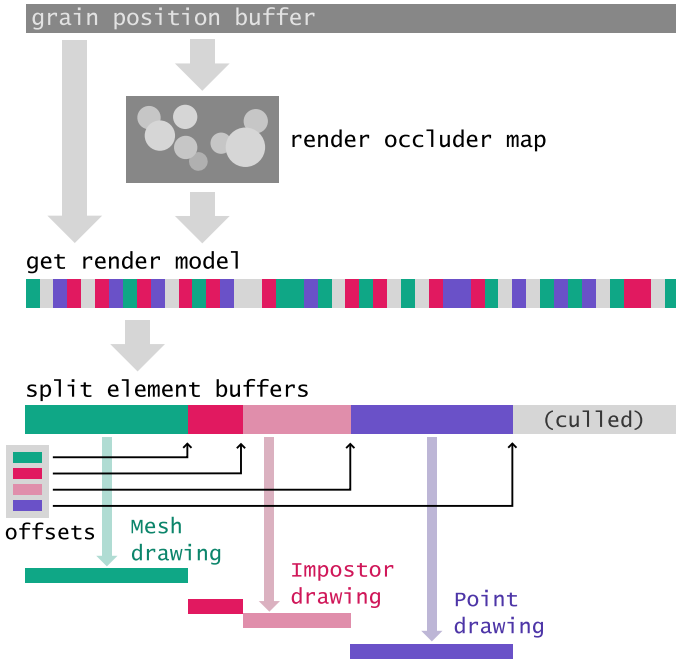


Figure V.10: ***Grain splitting.*** *Given the position of the grains, we first render a map of the most likely occluder grains and then distinguishes which model to use for each grain, building contiguous element buffers for each subsequent draw call. Impostor rendering requires two element buffers to render occluder candidates first (Section V.3.5.2).*

The splitting process consists in building those elements buffers from the array of all stacked elements. The number of elements being by hypothesis very large, it is not possible to pay for the round trip to the CPU, hence we perform this splitting entirely on the GPU, in compute shaders. It is also not even possible to sort the whole buffer by the distance of the grains to the view point.

Even if the splitting apparently distinguishes between only three models, it is more convenient to see it as operating on an arbitrary m number of models because next sections will add an extra state for culled points (Section V.3.5.1) and then split the impostors into two arrays for more efficient rendering (Section V.3.5.2).

We assume that we have a function `uint getRenderModel(uint element)` that returns for an element index the index from 0 to $m - 1$ of the model that must be used to draw it. This basically fetches the position buffer to check the distance to the grain against the thresholds, and will later on include culling. The output element arrays are written next to each other in a buffer allocated with the same size as the input element array. Besides this output, the methods returns a list of m offsets within the buffer to tell at which index each element array starts (Figure V.10).

Global atomic splitting We adopt a simple and effective method made of two steps. First, we atomically count the number of elements per model, in order to determine the output offsets. In a second step, we insert element indices in the output using for each model a second counter besides the offset to keep track of where is the next available index. This counter is atomically incremented each time an element is written. This process requires calling `getRenderModel` twice for each grain. Although this function gets more complex when culling is added, caching its output between the first and the second steps saves only a few tenths of millisecond on a stacking of 1.6M elements which is not worth the overhead of allocating a cache buffer. Once the element buffers are ready, the offsets can be used to build a command buffer adapted to each model in a simple compute buffer.

Scalability At this point, we have a pipeline able to render stackings of which grains can smoothly turn from meshes to points. But, as we intend to draw a large number of grains, we need to improve the scalability of the pipeline. Indeed, the use of impostors make the fragment processing even heavier than it usually tends to be in modern engines, so in the next sections we make use of the relative density of the stacking to **(i)** reduce the number of points emitting fragments (Section V.3.5.1) and **(ii)** reduce the number of emitted fragments that reach the fragment shader (Section V.3.5.2).

V.3.5 Occlusion Culling

V.3.5.1 Grain-level culling

In a dense stacking, a large proportion of the elements is totally invisible. We propose in this section a novel occlusion culling that is conservative, i.e., it does not cull visible objects, and based on the quasi-spherical proxy assumption. The occluder map it computes is further reused to improve per-fragment occlusion culling (Section V.3.5.2).

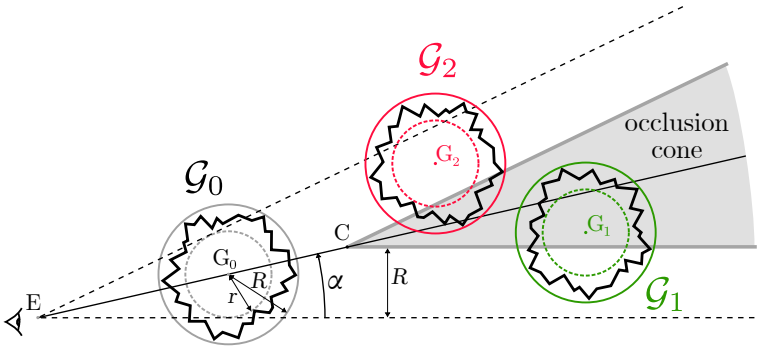


Figure V.11: *Since a grain is bounded by two spheres of radius r and R, when the center of a grain $\mathcal{G}_1$ lies in the occlusion cone, it is fully occluded by $\mathcal{G}_0$. On the contrary, the grain $\mathcal{G}_2$ might not be totally hidden and cannot culled out.*

Occlusion culling operates before the actual shape of grains is known, but can use the quasi-spherical proxy to early detect occlusions. If the inner sphere of a grain totally hides the outer sphere of another one, then no matter their actual shapes the second one will never be visible and can hence be safely culled out. As illustrated in Figure V.11, the inner sphere of a grain close to the view point creates a cone of occluded positions (dashed lines). This cone is eroded with the outer sphere to give a set of grain centers that can be culled (occlusion cone). Yet, it is far too expensive to test every pair of grains for occlusion. And while in theory this could be executed using hardware occlusion queries (Sekulic, 2004), with the inner sphere being the occluder and the outer sphere the proxy, it is not practical as it would require to render grains sequentially.

Occluder map Instead, we test each grain – the *occludee* – against exactly one other grain that we chose carefully – the *occluder candidate*. A grain can hide another one if it is at the same time closer to the view point, and projects around the same pixel on screen. So, prior to the occlusion test, we render the whole point cloud a first time. The z-test ensures that we keep the closest grain for each pixel i.e., the most likely to hide other grains. At this stage, no attribute fetching or computation is executed, instead the framebuffer is filled with the occluder candidate parameters – one occluder per pixel. These parameters are the position and radius of the occluder's inner sphere, fitting in a standard four-component color attachment.

To fill this *occluder map*, one must compute the point sizes: drawing points of exactly one pixel each would mean that the occluder candidate of a point is always the grain that projects on the same pixel but is closer to the camera. This has perfect chances of picking the right occluder candidate when it finds one, but will most of the time not find any other occluder candidate than the grain itself. On the other side, drawing the points using their inner radius is not the best choice either, because it will too often suggest an occluder candidate that is actually not occluding the point. Our trade-off is to render points large enough for all pixels to be covered by a few fragments while remaining as small as possible. In practice, for as dense as possible stackings viewed at distance for which impostors are used, we found experimentally that optimal values are located between 0.15 to 0.20 times the inner radius r.

Splitting Once this occluder map has been generated, the discrimination function `getRenderModel` in the splitting shader computes the screen pixel onto which a grain's center gets projected, and samples the occluder map at this coordinate. This gives the parameters of an occluder to test the current point against using the procedure detailed in Listing V.2. If the point is inside the occlusion cone, the function returns an index corresponding to no model.

```
1 bool IsOccluded(vec3 g1, mat4 proj, sampler2D occMap) {
2   vec4 clip = proj * vec4(g1, 1.0);
```

```
  vec4 occ = texture(occMap,clip.xy/clip.w*.5+.5);
  if (occ == NONE) return false;
  vec3 g0 = occ.xyz;
  float r = occ.a;
  float cosBeta = dot(
      normalize(g0),
      normalize(g1 - g0 * R / r)
  );
  if (cosBeta < 0) return false;
  float sinAlpha = r / length(g0);
  float sin2Beta = 1. - cosBeta * cosBeta;
  float sin2Alpha = sinAlpha * sinAlpha;
  return sin2Beta < sin2Alpha;
}
```

Listing V.2: *Returns true if the grain at position `g1` is occluded, given an occluder map rendered using the same projection matrix as the current view. This map contains the position `g0` and inner radius `r` of another grain or a mock value `NONE` (used to clear the buffer before rendering the map). Coordinates are in camera space.*

V.3.5.2 Fragment-level culling

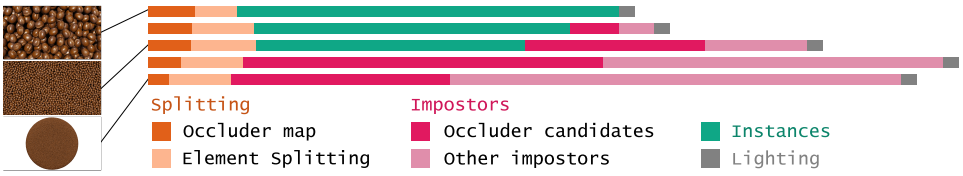


Figure V.12: *Breakdown of several frames' draw sequence during a reference shot from tight to large view over a stack of 1.6M coffee beans. On the left-hand side are (top-down): first frame, middle frame and last frame. These results focus on the transition from meshes to impostors.*

Sampling an impostor's maps is a costly operation, both in terms of memory bandwidth and computing power. There are two ways to reject a fragment before it reaches the fragment shader. One rather drastic is to discard the whole point, this was the goal of Section V.3.5.1. But this is not enough. For many grains beyond the first layer, only a couple of fragments are visible out of the tens or hundreds that it may cover. We still end up with hundreds of millions of fragments to shade, so we leverage another mechanism: *Early-Z Rejection*.

Visibility These hundreds of millions of fragments outnumber by several orders of magnitude the pixel count of a typical HD render (2M pixels) or even a 4K render (8M pixels), so there is mechanically a large proportion of *wasted fragments*, i.e., fragments that reach the fragment shader but are ultimately not visible on screen. This phenomenon is commonly referred as *overdraw*.

Over-shading is not specific to impostors, it is actually the prior motivation of deferred shading. But the core difficulty that impostor rendering introduces is the impossibility to determine the visibility of a fragment before sampling the maps. This deferred shape evaluation prevents us from using strategies such as visibility buffering (Burns & Hunt, 2013).

Early-Z Rejection The early-Z rejection is automatically performed by modern GPU's rendering pipelines (Sekulic, 2004). If a fragment lies behind the one already stored in the output buffer, then it can be rejected without being processed, provided that the shader does not override fragment's depth. Thus the benefits of early-Z rejection depend on the order in which points are rendered, and we have too many points to sort them front to back. Nevertheless, what early-Z rejection tells us is that the visibility does not need to be perfectly solved in order to gain in efficiency. We can split the grains into the *likely visible* ones and the *likely hidden* ones, and render the former first. This first draw call fills almost all pixels with their final value, so the second one sees most of its fragments early rejected. This is referred to below as the *double draw* scheme.

Implementation Fortunately, the question of determining likely visible grains has already been answered: those are the occluder candidates of the occlusion culling step. They represent a thin shell of closer grains for which most of the fragments are visible. In practice, we make the splitter distinguish separate elements buffers for occluder candidates and remaining points. When rendering impostor, the same draw call is repeated twice with these different element arrays. This simple change brings a significant speed-up to the overall impostor rendering.

V.3.6 Results

The performance of our C++/OpenGL implementation has been measured on an Nvidia GeFroce GTX 1070 graphics chip with 8GB of VRAM, on frames of 1920×1080 pixels. We focus the performance tests on the transition from impostors to meshes, where it is the most critical. We compare our impostor cloud at different angular resolutions to instances of the original grain mesh or a simplified mesh.

Breakdown The overall render time of a frame is subject to various factors. First, it varies significantly with the view point. In order to grasp the benefits of our method on real case scenarios, we evaluated performance during a backward dolly shot, from tight to large. Left-hand side of Figure V.12 shows first, middle and last frames of this test shot and breakdowns of these key frames. This qualitative evaluation already highlight a few points. First, although it is not negligible, the splitting process is not the bottleneck. Second, occluder map render time increases as point size grows on screen. Third, the Z-prepass does not have a significant impact as this draw call involves almost no fragment processing. Fourth, the core element splitting is rather constant, until most grains get frustum culled in closer

Figure V.13: *Impostor clouds built from diverse grain models. Impostors use 128 precomputed views ($n = 8$) of 128×128 pixels each.*

views. Last, despite being unbalanced in number of points, the first and second draw calls of impostor rendering takes similar times. This is satisfactory as it suggests that we found a reasonable trade-off between rendering a few costly points first and then more points but which are less visible.

Performance This high variability of draw mixture within a single frame makes it hard to draw proper conclusions, so in Figure V.14 we compare scenarios without splitting, where only one of meshes or impostor models is used. The timings for impostor rendering do not depend on the original complexity of the grain, so we compare them to several meshes. A first thing to notice is that we indeed need a hybrid model since when the number of grains within the view frustum is low (close viewpoint) instanced meshes are more efficient than impostors while as it increases impostors eventually outperform instances.

The shape of instance and impostor curves are different because the former is

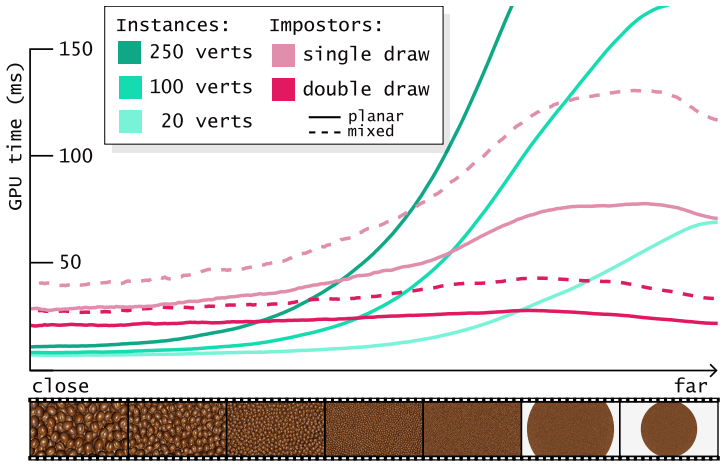


Figure V.14: *Render time on a scene made of 1.6M grains. Thumbnails of the test sequence can be found bellow the horizontal axis. Impostors use 128 views of 128 pixels. Both instances and impostors use our occlusion culling method.*

more affected by the number of vertices to draw (vertex bounded) while the latter is related to the number of pixels (pixel bounded). In case of a perfectly pixel bounded rendering, our test shot should take a constant render time. The results of Figure V.14 show that it is not the case when naively drawing all the impostors at once (*single draw*). This is because of the large number of overdrawn fragments. Our double draw scheme on the other hand succeeds at reducing overdraw, as shown by its more constant render time. It thus makes our mixed sampling competitive despite its overhead.

As discussed in Section V.3.4.1, visual accuracy sets a minimal distance at which transitioning from meshes to impostors. These results show that when the grains have shapes requiring a low amount of vertices, pushing this threshold distance further can increase performance. For more complex grains, the threshold is already beyond the cross point between green and red lines so there is no interest in increasing it. Even when combining our approach with usual mesh LoD, the vertex count does not reduce beyond a few tens, so an eventual switch to impostors is beneficial.

Visual loss Figures V.13 shows impostor clouds of twenty thousand points at different scales and in different scenarios, illustrating the variety of possible grain shapes. Figure V.3 is a more extreme example featuring two million grains. To evaluate the visual loss of our model, we measured the structural similarity (SSIM) between animations rendered using different impostors on one hand and a reference render using meshes on another hand. Figure V.16 compares variations of the choice of sampling scheme at fixed memory use with variations of the number of stored precomputed views. The stacked grain used for this example is the coffee

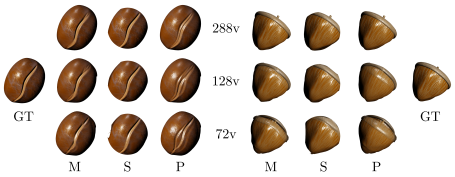


Figure V.15: *Impostors rendered using mixed (M), spherical (S) or planar (P) samplings with various number of pre-computed views, along with ground truth (GT).*

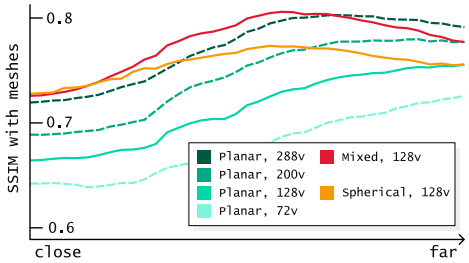


Figure V.16: *SSIM measures using different sampling schemes on the coffee bean: planar, spherical and mixed (ours).*

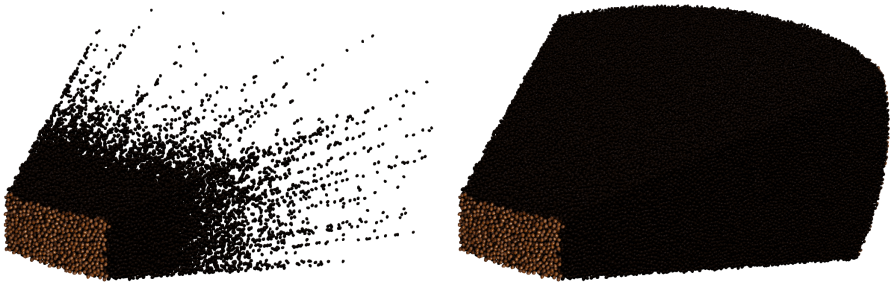


Figure V.17: *View frustum with (left) and without (right) our grain occlusion culling. Some of the remaining points may actually be hidden, but it is ensured that no visible point is removed.*

bean of Figure V.15, left. We see that for equivalent memory requirements, our mixed sampling gives better visual accuracy.

Occlusion culling Figure V.17 illustrates the effect of our occlusion culling on a dense volume of grains. As shown by the graph of Figure V.18, the ability of our method to cull grains decreases progressively as we relax the hypothesis of a non

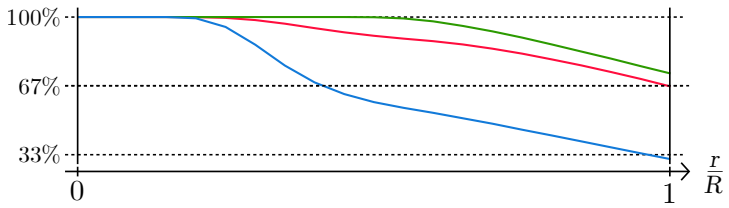


Figure V.18: *Proportion of grains still rendered after the occlusion culling step, depending on the inner over outer radius ratio on three different shots. Variations among shots are due for instance to a larger foreground for the blue curve.*

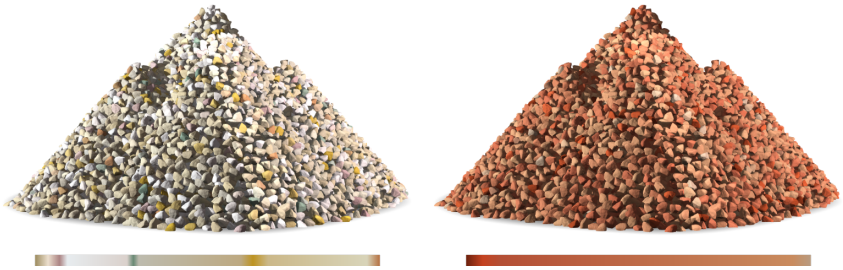


Figure V.19: *Impostor cloud where precomputed attributes such as normal vectors are used along with dynamic procedural attributes such as albedo values, which is drawn from the color ramp underneath each image.*

null inner radius r. The effect of the culling varies with the frame. The green curve was measured with a narrower field of view. The camera rays are in that case more parallel, hence there is less occlusion detected with our method. The blue curve was measured on a more favorable scenario, where grains in the foreground hide significant parts of the whole set.

V.3.7 Discussion

V.3.7.1 Properties

Our methods can render stackings of millions of dynamic objects in real time, leveraging the similarity of dense quasi-spherical grains using impostors to design a transition LoD which provides an efficient trade off – both in terms of accuracy and speed – when individual grains only cover a few pixels on the screen. This is tracktable thanks to our new sampling scheme that reduces memory usage for a given visual loss, together with a coupled per-grain/fragment occlusion mechanism. Contrary to instancing, the complexity of the impostors is independent on the original model. Thanks to our occluder map and splitting scheme, it is mostly dependent on the output resolution. Our method is compatible with arbitrary animation of the grain positions, which is important in an authoring pipeline, and scales to large stackings (Figure V.20).

Being designed to feed the G-pass of a deferred shading engine, dynamic procedural variations of the grain can be coupled with the precomputed data at render time (Figure V.19), reducing further potential repetition effects while expanding visual diversity. Our approach can easily be integrated to a modern render pipeline based on PBR materials and deferred shading, shadow mapping and more.

Graceful degradation Moreover, our method degrades gracefully regarding all of its hypotheses (quasi-spherical grains, density, moderate shape diversity), either in accuracy or in efficiency depending on the application context. We sacrifice accuracy during an interactive authoring session, but the *look dev* artists using the

shape in their scene have the possibility to change the trade-off at any time.

V.3.7.2 Limitations

Grain shape For non quasi-spherical enough grains, visual loss must be balanced with more precomputed views. At some point, the hypothesis of quasi-spherical shape made by our mixed sampling scheme becomes as invalid as using planar sampling. An extreme example that breaks our hypothesis is a tubular element, e.g., a threaded nut.

Self-intersection Also, grains must not intersect each other. We do not change the fragment depth when rendering them, so they are sorted by the depth of their center, provoking popping artifacts in case of intersection. Writing a precomputed depth in the Z-buffer when rendering the impostor is possible, but at the expense of important performance reduction because this would turn off early-Z rejection. Another consequence of this per-grain depth is that the standard shadow maps cannot render grains' self-shadows.

Far grains The rendering model that we used to render far grains beyond the validity range of our impostor is subject to aliasing. To improve the transition from impostors to pure point based rendering, more advanced existing point-based models could be used. Note that we did not chose to switch to a surface-based representation, such as Bruneton and Neyret (2012) do, because we did not want to give up on the ability to animate grains.

V.3.7.3 Future work

Our method can be further developed along several directions. First, we could also defer the sampling of attributes in a separate pass to save memory bandwidth. Impostor rendering would query only the alpha channel, to build a visibility buffer (Burns & Hunt, 2013). We could accumulate fragments during the first of the two draw calls. This would enable cross grain alpha blending and hence reduce aliasing when grains come very close to being points. Accumulation disables the benefits of early-Z rejection but it is mostly the second pass that benefits from it.

Second, our memory usage can be further improved. We do not use any form of texture compression and use heavy 32-bit color attachments. Also, a grain being bounded by a sphere, the corners of precomputed views are always left unused. Todt et al. (2007) use a distorted mapping to address this issue, but this prevents us from using standard mipmaps. Furthermore, the attribute field captured by our rich impostors have a lower dimensionality than the light field captured by radiance impostors. Hence, there is more redundancy in our representation, that could be compressed better, storing only the mapping from position on the bounding sphere and ray orientation to UV space, which yields interesting filtering issues to address.

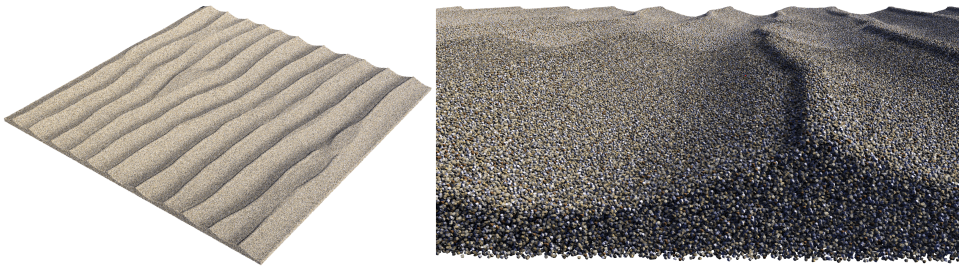


Figure V.20: *A large dynamic scene made of 20M sand grains and rendered with our method in 56ms on a GeForce 1070 GPU.*

More generally, we would like to study the possibility for shape programs to automatically generate multiple LoD of their output. This goes together with closing the gap between pure geometry and appearance modeling. Both materials and shapes happen to be represented by programs, but the potential for transferring data from one to the other, similarly to what impostors do, remains under explored.

VI

Conclusion

We opened this thesis by advocating the use and analysis of higher-order representations of shapes, in particular program-based representations. They enable a renewal of 3D digital creation workflows, where artistic decision taking can be postponed to mitigate the cost of creation loops. We have explored the diversity of such representations, from imperative DAGs to declarative tiling through hybrid paradigms like parametric tiles, addressing the overall question of how to alleviate the frictions of shape programming.

VI.1 Contributions

We showed that program-based representations of shapes embed meaningful information about the intent of their designer, and that we can practically leverage it to assist the creation process. For imperative programs, represented as DAGs, we capture this semantic information through the automatic on-the-fly injection of new nodes, what we call *DAG amendment*. These nodes aim at augmenting the output geometry with application-specific labels: a *co-parameter* when we need to recognize points upon change of hyper-parameters in order to compute differential information, or a *trace* of operations, when we want to characterize the role that geometry elements are playing in a DAG.

Both of these amendments have applications that address core challenges of program-based representations. Our co-parameterization enables the direct manipulation of a shape program's output in cases that are not handled by typical inverse kinematics techniques, in particular when the connectivity of the output mesh fluctuates. And we coupled our trace recording with program synthesis methods to assist the creation of parts of a shape program responsible for selecting geometry, namely *selection queries*. Thus, DAG amendments not only ease the tuning of a program's input hyper-parameters, but they also companion the upstream design

of this program.

In the case of declarative programs, some of the designer's intent is made explicit by layout rules, e.g., the interface labeling, for Wang tiles. We used these rules to constrain the geometrical content of the tiles in our mesostructure generation system, adopting an *interface-centric* approach to ensure by construction the continuity of the program's output.

The composition of a declarative program is a dialog with a solving engine. We make this dialog more fluent by having the engine suggest new tiles to add, to help the designer pave a target domain. We make this dialog more versatile by extending the usual discrete Wang tiling problem to continuous sets of tiles and interfaces. We thus enable a new hybrid shape programming paradigm, that mixes both a tiling system and parametric shapes to define the content of so-called *parametric tiles*.

We integrated shape programs with real-time rendering pipelines. This synergy is at the same time needed and enabled by program-based representation. Needed, because a key application of shape programs is the conception of authoring workflows, and enabled, thanks the structural information that a shape program provides. For two examples of shape programs generating heavy geometrical content, we deferred the evaluation of some parts of the shape and off-loaded it on the GPU, and we altered the program to dynamically adapt to the view point and output different first-order representations of the same shape.

Through multiple fully functional authoring and rendering systems, we showed how representing and manipulating shapes as higher-order programs rather than first-order geometry enables new creation workflows, leading to digital assets as rich and versatile as parametric shapes.

VI.2 Future prospects

In a way, we have only scratched the surface of the challenges introduced in Section I.2.4. Each one of these key directions can be further explored in the continuity of our approaches.

Ensuring generalization Ensuring generalization means to transform instance-specific user gestures into programs that capture the intent behind these gestures. We addressed the case of selection gestures and replaced instance-specific lists of indices with symbolic queries which generalize much better. This use of program synthesis within the creation loop can be ported to other gestures, one being the computation of unexposed node parameters. For instance when the designer rotates a part of an object by 40°, was in order to point it to another one? Or to align it to its neighbors? This problem is less constrained than the selection, for which we could consider each element of geometry as a different example, so it

requires either more prior knowledge, or more user input. Or a different type of user interaction, where the tool engages a proper dialog with the designer by asking for disambiguation.

Assisted authoring of shape programs The dialog-based approach was the spirit behind our tile suggestion mechanism. As we highlighted, it is particularly needed for declarative programming of shapes in general: the solving engine generally faces over or under-constrained problems about which it must be transparent enough if we want to integrate them into an interactive creation tool. We focused on tiled layout, but other layout engines, e.g., more focused on alignment, distribution and other 3D kitbashing operations, could also be used for shape programming. One may also develop new paradigms, like we have initiated with our parametric tile engine.

Program synthesis can be used together with shape analysis in order to develop workflows where program-based representations are used in cooperation with other sources of 3D data such as 3D scanning. For instance, *InverseCSG* (2018) bridges program synthesis with RANSAC-based shape analysis; similarly symmetry detection and other dictionary-based analysis (Lescoat et al., 2018) can be a first step towards the automatic construction of shape programs.

Shape manipulation Our general strategy for extracting information from the structure of a shape program is to amend it with automatic rewriting rules so that it output extra details. This approach can be explored beyond our two examples. In both cases we need DAG nodes to be able to relate their output to their input: in a way nodes are augmented with extra methods. How else could we augment them? Nodes could provide information about the visibility or bounding box of their content. This is related to abstract interpretation, another branch of the programming language literature that we can enroll for the study of shape programs, together with program synthesis.

Integration with other programs The visibility-related static analysis of shape programs that we just mentioned could be leveraged to push further what we have done on particular cases regarding the integration of shape programs and real-time rendering. We focused on the instantiation of similar atoms, for which our prior on the boundary spheres or the number of sweep components is uniform, but we could look for more general program-level culling, using the program to generate a simplified geometry. In the case of grain rendering, we accounted for the role of materials. Materials are sometimes themselves generated by image processing DAGs; could we have these DAG interact with each others? Using height maps, it is common that users of *Adobe Substance Designer* embed a lot of geometrical information in the procedural materials they build.

Although we were primarily interested in rendering in the context of authoring

tools, similar approaches might be used to create optimized game-ready parametric assets, or to synthesize acceleration structures for offline rendering. Or even acceleration structures for other kinds of physic simulations. The additional information contained in the structure of a program can serve as a prior for some under-constrained algorithms.

Shape IDE Holding all these challenges together within a consistent toolkit for visual programming of shapes corresponds in the end to what generic (text-based) programming calls an Integrated Development Environment. What would such a *shape IDE* look like? Providing tools for prototyping, debugging, refactoring, optimizing shape programs makes as much sense as for regular programs. And the specific context of shape programming suggests novel tools, like bidirectional editing, which couple symbolic and spatial manipulations of shapes.

Symbolic AI Our last point is that studying shape programs fosters the development of symbolic artificial intelligence for computer graphics. We have witnessed in the recent years a growing use of AI techniques in computer graphics, but this use is so far mainly focused on continuous AI, namely machine learning, and in particular deep learning. There is nevertheless a lot of potential for symbolic AI like program synthesis, and its emergence necessarily goes through the use of symbolic, i.e., program-based, representations of shapes. Symbolic AI is particularly relevant in the context of collaboration between a machine and a human for the authoring of creative assets, because we need machine-generated models to be human-understandable and manipulable.

We hope that our work on program-based representations of 3D shapes will contribute to the development of collaborations between programming languages, artificial intelligence, human-computer interaction and computer graphics.

Bibliography

Abdrashitov, R., Chevalier, F., & Singh, K. (2020, October). Interactive Exploration and Refinement of Facial Expression using Manifold Learning. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology* (pp. 778–790). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3379337.3415877

Agbodan, D., Marcheix, D., & Pierra, G. (2000). Persistent Naming for Parametric Models. In *Proceedings of WSCG '2000* (pp. 418–425). University of West Bohemia.

Albarghouthi, A., Gulwani, S., & Kincaid, Z. (2013). Recursive Program Synthesis. In N. Sharygina & H. Veith (Eds.), *Computer Aided Verification* (pp. 934–950). Berlin, Heidelberg: Springer. doi: 10.1007/978-3-642-39799-8_67

Ali-Hamadi, D., Liu, T., Gilles, B., Kavan, L., Faure, F., Palombi, O., & Cani, M.-P. (2013, November). Anatomy transfer. *ACM Trans. Graph.*, *32*(6). doi: 10.1145/2508363.2508415

Aliaga, D. G., Demir, İ., Benes, B., & Wand, M. (2016, July). Inverse procedural modeling of 3D models for virtual worlds. In *ACM SIGGRAPH 2016 Courses* (pp. 1–316). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2897826.2927323

Aliaga, D. G., Vanegas, C. A., & Benes, B. (2008, December). Interactive example-based urban layout synthesis. *ACM Transactions on Graphics*, *27*(5), 160:1–160:10. doi: 10.1145/1409060.1409113

Alur, R., Černý, P., & Radhakrishna, A. (2015). Synthesis Through Unification. In D. Kroening & C. S. Păsăreanu (Eds.), *Computer Aided Verification* (pp. 163–179). Cham: Springer International Publishing. doi: 10.1007/978-3-319-21668-3_10

Anguelov, D., Srinivasan, P., Koller, D., Thrun, S., Rodgers, J., & Davis, J. (2005).

SCAPE: Shape completion and animation of people. In *ACM SIGGRAPH 2005 papers* (pp. 408–416). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1186822.1073207

Angus Johnson. (2014, January). *Clipper.*

Aristidou, A., Lasenby, J., Chrysanthou, Y., & Shamir, A. (2018). Inverse Kinematics Techniques in Computer Graphics: A Survey. *Computer Graphics Forum*, *37*(6), 35–58. doi: 10.1111/cgf.13310

Association, T. O. (2006). *OpenFX.* https://openfx.readthedocs.io.

Au, O. K.-C., Tai, C.-L., Chu, H.-K., Cohen-Or, D., & Lee, T.-Y. (2008, August). Skeleton extraction by mesh contraction. *ACM Trans. Graph.*, *27*(3), 1–10. doi: 10.1145/1360612.1360643

Aujay, G., Hétroy, F., Lazarus, F., & Depraz, C. (2007, August). Harmonic skeleton for realistic character animation. In M. Gleicher & D. Thalmann (Eds.), *SCA '07 - ACM-SIGGRAPH/Eurographics symposium on computer animation* (pp. 151–160). San Diego, United States: Eurographics Association. doi: 10.2312/SCA/SCA07/151-160

Avril, Q., Ghafourzadeh, D., Ramachandran, S., Fallahdoust, S., Ribet, S., Dionne, O., … Paquette, E. (2016). Animation setup transfer for 3D characters. *Computer Graphics Forum*, *35*(2), 115–126. doi: 10.1111/cgf.12816

Baerlocher, P., & Boulic, R. (1998, October). Task-priority formulations for the kinematic control of highly redundant articulated structures. In *Proceedings. 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems. Innovations in Theory, Practice and Applications (Cat. No.98CH36190)* (Vol. 1, p. 323-329 vol.1). doi: 10.1109/IROS.1998.724639

Baerlocher, P., & Boulic, R. (2004). An inverse kinematics architecture enforcing an arbitrary number of strict priority levels. *The visual computer*, *20*(6), 402–417.

Bailey, S. W., Omens, D., Dilorenzo, P., & O'Brien, J. F. (2020, July). Fast and deep facial deformations. *ACM Transactions on Graphics*, *39*(4), 94:94:1–94:94:15. doi: 10.1145/3386569.3392397

Bailey, S. W., Otte, D., Dilorenzo, P., & O'Brien, J. F. (2018, July). Fast and deep deformation approximations. *ACM Transactions on Graphics*, *37*(4), 119:1–119:12. doi: 10.1145/3197517.3201300

Baran, I., & Popović, J. (2007). Automatic rigging and animation of 3D characters. In *ACM SIGGRAPH 2007 papers.* New York, NY, USA: ACM. doi: 10.1145/1275808.1276467

Barroso, S., Besuievsky, G., & Patow, G. (2013). Visual copy & paste for procedurally modeled buildings by ruleset rewriting. *Computers & Graphics*, *37*(4), 238–246. doi: 10.1016/j.cag.2013.01.003

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2018). Automatic Differentiation in Machine Learning: A Survey. *Journal of Machine Learning Research*, *18*(153), 1–43.

Beaudouin-Lafon, M. (2000, April). Instrumental interaction: An interaction model for designing post-WIMP user interfaces. In *Proceedings of the SIGCHI*

conference on Human Factors in Computing Systems (pp. 446–453). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/332040.332473

Beaudouin-Lafon, M., Bødker, S., & Mackay, W. E. (2021, November). Generative Theories of Interaction. *ACM Transactions on Computer-Human Interaction*, *28*(6), 45:1–45:54. doi: 10.1145/3468505

Belhadj, F., & Audibert, P. (2005, November). Modeling landscapes with ridges and rivers: Bottom up approach. In *Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia* (pp. 447–450). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1101389.1101479

Beneš, B., & Forsbach, R. (2002). Visual simulation of hydraulic erosion. *Journal of WSCG*, *10*(1-2), 79–86.

Bergroth, L., Hakonen, H., & Raita, T. (2000, September). A survey of longest common subsequence algorithms. In *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000* (pp. 39–48). doi: 10.1109/SPIRE.2000.878178

Bhat, P., Ingram, S., & Turk, G. (2004, July). Geometric texture synthesis by example. In *Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing* (pp. 41–44). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1057432.1057437

Bian, X., Wei, L.-Y., & Lefebvre, S. (2018). Tile-based pattern design with topology control. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, *1*, 23–38. doi: 10.1145/3203204

Blanz, V., & Vetter, T. (1999). A morphable model for the synthesis of 3D faces. In *Proceedings of the 26th annual conference on computer graphics and interactive techniques* (pp. 187–194). USA: ACM Press/Addison-Wesley Publishing Co. doi: 10.1145/311535.311556

Bokeloh, M., Wand, M., Seidel, H.-P., & Koltun, V. (2012, July). An algebraic model for parameterized shape editing. *ACM Transactions on Graphics*, *31*(4), 78:1–78:10. doi: 10.1145/2185520.2185574

Botsch, M., Pauly, M., Gross, M. H., & Kobbelt, L. (2006). PriMo: Coupled prisms for intuitive surface modeling. In *Symposium on geometry processing* (pp. 11–20).

Botsch, M., & Sorkine, O. (2008, January). On Linear Variational Surface Deformation Methods. *IEEE Transactions on Visualization and Computer Graphics*, *14*(1), 213–230. doi: 10.1109/TVCG.2007.1054

Brainerd, W., Foley, T., Kraemer, M., Moreton, H., & Nießner, M. (2016, July). Efficient GPU rendering of subdivision surfaces using adaptive quadtrees. *ACM Transactions on Graphics*, *35*(4), 1–12. doi: 10.1145/2897824.2925874

Brodersen, A., Museth, K., Porumbescu, S., & Budge, B. (2008, March). Geometric Texturing Using Level Sets. *IEEE Transactions on Visualization and Computer Graphics*, *14*(2), 277–288. doi: 10.1109/TVCG.2007.70408

Brucks, R. (2018). *Octahedral impostors.* https://shaderbits.com/blog/octahedral-impostors.

Bruneton, E., & Neyret, F. (2012). Real-time realistic rendering and lighting of forests. *Computer Graphics Forum*, *31*(2pt1), 373–382.

Bunel, R., Hausknecht, M., Devlin, J., Singh, R., & Kohli, P. (2018, February). Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis. In *International Conference on Learning Representations*.

Burnett, M. M. (1999). Visual Programming. In *Wiley Encyclopedia of Electrical and Electronics Engineering*. John Wiley & Sons, Ltd. doi: 10.1002/047134608X.W1707

Burns, C. A., & Hunt, W. A. (2013, August). The visibility buffer: A cache-friendly approach to deferred shading. *Journal of Computer Graphics Techniques (JCGT)*, *2*(2), 55–69.

Capell, S., Burkhart, M., Curless, B., Duchamp, T., & Popović, Z. (2005). Physically based rigging for deformable characters. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on computer animation* (pp. 301–310). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1073368.1073412

Capoyleas, V., Chen, X., & M Hoffmann, C. (1996, January). Generic naming in generative, constraint-based design. *Computer-Aided Design*, *28*(1), 17–26. doi: 10.1016/0010-4485(95)00014-3

Cascaval, D., Shalah, M., Quinn, P., Bodik, R., Agrawala, M., & Schulz, A. (2022). Differentiable 3D CAD programs for bidirectional editing. *Computer Graphics Forum*, *40*(2). doi: 10.1111/cgf.14476

Chaudhuri, S., Kalogerakis, E., Giguere, S., & Funkhouser, T. (2013, October). Attribit: Content creation with semantic attributes. In *Proceedings of the 26th annual ACM symposium on User interface software and technology* (pp. 193–202). St. Andrews Scotland, United Kingdom: ACM. doi: 10.1145/2501988.2502008

Chen, X., & Hoffmann, C. M. (1995, September). Towards feature attachment. *Computer-Aided Design*, *27*(9), 695–702. doi: 10.1016/0010-4485(94)00027-B

Chen, Z., Tagliasacchi, A., & Zhang, H. (2020). BSP-Net: Generating Compact Meshes via Binary Space Partitioning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 45–54).

Chen, Z., & Zhang, H. (2019). Learning Implicit Fields for Generative Shape Modeling. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 5939–5948).

Chiu, C.-H., Koyama, Y., Lai, Y.-C., Igarashi, T., & Yue, Y. (2020, July). Human-in-the-loop differential subspace search in high-dimensional latent space. *ACM Trans. Graph.*, *39*(4). doi: 10.1145/3386569.3392409

Chomsky, N. (1956, September). Three models for the description of language. *IRE Transactions on Information Theory*, *2*(3), 113–124. doi: 10.1109/TIT.1956.1056813

Chugh, R. (2016, May). Prodirect manipulation: Bidirectional programming for

the masses. In *Proceedings of the 38th International Conference on Software Engineering Companion* (pp. 781–784). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2889160.2889210

Chugh, R., Hempel, B., Spradlin, M., & Albers, J. (2016, June). Programmatic and direct manipulation, together at last. *ACM SIGPLAN Notices*, *51*(6), 341–354. doi: 10.1145/2980983.2908103

Ciolfi Felice, M., Maudet, N., Mackay, W. E., & Beaudouin-Lafon, M. (2016, October). Beyond Snapping: Persistent, Tweakable Alignment and Distribution with StickyLines. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (pp. 133–144). Tokyo Japan: ACM. doi: 10.1145/2984511.2984577

Cohen, M. F., Shade, J., Hiller, S., & Deussen, O. (2003, July). Wang Tiles for image and texture generation. *ACM Transactions on Graphics*, *22*(3), 287–294. doi: 10.1145/882262.882265

Cook, R. L. (1984, January). Shade trees. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (pp. 223–231). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/800031.808602

Cook, R. L., Halstead, J., Planck, M., & Ryu, D. (2007). Stochastic simplification of aggregate detail. *ACM Trans. Graph.*, *26*(3).

Cordonnier, G. (2018). *Layered Models for Large Scale Time-Evolving Landscapes* (Unpublished doctoral dissertation). Université Grenoble Alpes.

Cordonnier, G., Cani, M.-P., Benes, B., Braun, J., & Galin, E. (2018, May). Sculpting Mountains: Interactive Terrain Modeling Based on Subsurface Geology. *IEEE Transactions on Visualization and Computer Graphics*, *24*(5), 1756–1769. doi: 10.1109/TVCG.2017.2689022

Crassin, C., Neyret, F., Lefebvre, S., & Eisemann, E. (2009). GigaVoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on interactive 3D graphics and games* (pp. 15–22). New York, NY, USA: ACM. doi: 10.1145/1507149.1507152

Dachsbacher, C., Vogelgsang, C., & Stamminger, M. (2003, July). Sequential point trees. *ACM Trans. Graph.*, *22*(3), 657–662. doi: 10.1145/882262.882321

de Reffye, P., Edelin, C., Françon, J., Jaeger, M., & Puech, C. (1988, June). Plant models faithful to botanical structure and development. *ACM SIGGRAPH Computer Graphics*, *22*(4), 151–158. doi: 10.1145/378456.378505

Debevec, P. D., Taylor, C. J., & Malik, J. (1996). Modeling and rendering architecture from photographs: A hybrid geometry-and image-based approach. In *Proceedings of the 23th annual conference on computer graphics and interactive techniques.*

Decaudin, P., & Neyret, F. (2004, June). Rendering Forest Scenes in Real-Time. In *EGSR04: 15th Eurographics Symposium on Rendering* (p. 93). Eurographics Association.

Décoret, X., Durand, F., Sillion, F. X., & Dorsey, J. (2002). *Billboard clouds* (Unpublished doctoral dissertation). INRIA.

Demir, İ., Aliaga, D. G., & Benes, B. (2016, October). Proceduralization for Editing

3D Architectural Models. In *2016 Fourth International Conference on 3D Vision (3DV)* (pp. 194–202). doi: 10.1109/3DV.2016.28

Deng, B., Genova, K., Yazdani, S., Bouaziz, S., Hinton, G., & Tagliasacchi, A. (2020). CvxNet: Learnable Convex Decomposition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 31–44).

Deo, A. S., & Walker, I. D. (1992, May). Robot subtask performance with singularity robustness using optimal damped least-squares. In *Proceedings 1992 IEEE International Conference on Robotics and Automation* (p. 434-441 vol.1). doi: 10.1109/ROBOT.1992.220301

De Toledo, R., Wang, B., & Lévy, B. (2008). Geometry Textures and Applications†. *Computer Graphics Forum*, *27*(8), 2053–2065. doi: 10.1111/j.1467-8659.2008.01185.x

Deussen, O., & Lintermann, B. (2005). *Digital Design of Nature: Computer Generated Plants and Organics*. Springer Science & Business Media.

diSessa, A. A., & Abelson, H. (1986, September). Boxer: A reconstructible computational medium. *Communications of the ACM*, *29*(9), 859–868. doi: 10.1145/6592.6595

Douze, M., Franco, J.-S., & Raffin, B. (2017, June). *QuickCSG: Fast arbitrary boolean combinations of n solids* (Research Report). Grenoble: Inria.

Du, T., Inala, J. P., Pu, Y., Spielberg, A., Schulz, A., Rus, D., … Matusik, W. (2018, December). InverseCSG: Automatic conversion of 3D models to CSG trees. *ACM Transactions on Graphics*, *37*(6), 213:1–213:16. doi: 10.1145/3272127.3275006

Dupuy, J., Heitz, E., Iehl, J.-C., Poulin, P., Neyret, F., & Ostromoukhov, V. (2013). Linear efficient antialiased displacement and reflectance mapping. *ACM Transactions on Graphics (TOG)*, *32*(6), 211.

Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K., & Worley, S. (1994). *Texturing and Modeling: A Procedural Approach* (Bk&Disk edition ed.). Academic Press.

Efros, A. A., & Freeman, W. T. (2001, August). Image quilting for texture synthesis and transfer. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (pp. 341–346). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/383259.383296

Eisenberger, M., Novotny, D., Kerchenbaum, G., Labatut, P., Neverova, N., Cremers, D., & Vedaldi, A. (2021). NeuroMorph: Unsupervised Shape Interpolation and Correspondence in One Go. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 7473–7483).

Eitz, M., Richter, R., Boubekeur, T., Hildebrand, K., & Alexa, M. (2012, July). Sketch-based shape retrieval. *ACM Transactions on Graphics*, *31*(4), 31:1–31:10. doi: 10.1145/2185520.2185527

ElKoura, G., & Studios, P. A. (2013). Presto execution system: An asynchronous computation engine for animation. *Pixar Animation Studios*.

Ellis, T. O., Heafner, J. F., & Sibley, W. L. (1969, September). *The Grail Project: An Experiment in Man-Machine Communications* (Tech. Rep.). Santa Monica,

CA: Rand Corp.

Emilien, A. (2014). *Interactive design of virtual worlds : Combining procedural modeling with intuitive user control* (Unpublished doctoral dissertation). Université de Grenoble.

Emilien, A., Bernhardt, A., Peytavie, A., Cani, M.-P., & Galin, E. (2012, June). Procedural generation of villages on arbitrary terrains. *The Visual Computer*, *28*(6), 809–818. doi: 10.1007/s00371-012-0699-7

et al. Cignoni, P. (2008). MeshLab: An open-source mesh processing tool. In V. Scarano, R. D. Chiara, & U. Erra (Eds.), *Eurographics italian chapter conference.* The Eurographics Association.

Foster, J. N. (2009). *Bidirectional programming languages* (Unpublished doctoral dissertation). University of Pennsylvania.

Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., & Schmitt, A. (2007, May). Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, *29*(3), 17–es. doi: 10.1145/1232420.1232424

Gadelha, M., Gori, G., Ceylan, D., Mech, R., Carr, N., Boubekeur, T., … Maji, S. (2020). Learning Generative Models of Shape Handles. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 402–411).

Gaillard, M., Krs, V., Gori, G., Mech, R., & Benes, B. (2022). Automatic Differentiable Procedural Modeling. *Computer Graphics Forum*, *40*(2). doi: 10.1111/cgf.14475

Gaisbauer, W., Raffe, W. L., Garcia, J. A., & Hlavacs, H. (2019, October). Procedural Generation of Video Game Cities for Specific Video Game Genres Using WaveFunctionCollapse (WFC). In *Extended Abstracts of the Annual Symposium on Computer-Human Interaction in Play Companion Extended Abstracts* (pp. 397–404). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3341215.3356255

Gal, R., Sorkine, O., Mitra, N. J., & Cohen-Or, D. (2009). IWIRES: An analyze-and-edit approach to shape manipulation. In *ACM SIGGRAPH 2009 papers.* New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1576246.1531339

Galin, E., Guérin, E., Peytavie, A., Cordonnier, G., Cani, M.-P., Benes, B., & Gain, J. (2019, May). A Review of Digital Terrain Modeling. *Computer Graphics Forum*, *38*(2), 553–577. doi: 10.1111/cgf.13657

Galin, E., Peytavie, A., Maréchal, N., & Guérin, E. (2010). Procedural Generation of Roads. *Computer Graphics Forum*, *29*(2), 429–438. doi: 10.1111/j.1467-8659.2009.01612.x

Ganin, Y., Kulkarni, T., Babuschkin, I., Eslami, S. M. A., & Vinyals, O. (2018, July). Synthesizing Programs for Images using Reinforced Adversarial Learning. In *International Conference on Machine Learning* (pp. 1666–1675). PMLR.

Garland, M., & Heckbert, P. S. (1997). Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics*

and interactive techniques (pp. 209–216).

Génevaux, J.-D., Galin, É., Guérin, E., Peytavie, A., & Benes, B. (2013, July). Terrain generation using procedural models based on hydrology. *ACM Transactions on Graphics*, *32*(4), 143:1–143:13. doi: 10.1145/2461912.2461996

Génevaux, J.-D., Galin, E., Peytavie, A., Guérin, E., Briquet, C., Grosbellet, F., & Benes, B. (2015). Terrain Modelling from Feature Primitives. *Computer Graphics Forum*, *34*(6), 198–210. doi: 10.1111/cgf.12530

Genova, K., Cole, F., Sud, A., Sarna, A., & Funkhouser, T. (2020). Local Deep Implicit Functions for 3D Shape. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 4857–4866).

Genova, K., Cole, F., Vlasic, D., Sarna, A., Freeman, W. T., & Funkhouser, T. (2019). Learning Shape Templates With Structured Implicit Functions. In *Proceedings of the IEEE/CVF International Conference on Computer Vision* (pp. 7154–7164).

Germer, T., & Schwarz, M. (2009). Procedural Arrangement of Furniture for Real-Time Walkthroughs. *Computer Graphics Forum*, *28*(8), 2068–2078. doi: 10.1111/j.1467-8659.2009.01351.x

Girard, P. (2001, January). Chapter 7 - Bringing Programming by Demonstration to CAD Users. In H. Lieberman (Ed.), *Your Wish is My Command* (p. 135-VII). San Francisco: Morgan Kaufmann. doi: 10.1016/B978-155860688-3/50008-7

Girdhar, R., Fouhey, D. F., Rodriguez, M., & Gupta, A. (2016). Learning a Predictable and Generative Vector Representation for Objects. In B. Leibe, J. Matas, N. Sebe, & M. Welling (Eds.), *Computer Vision – ECCV 2016* (pp. 484–499). Cham: Springer International Publishing. doi: 10.1007/978-3-319-46466-4_29

Gobbetti, E., & Marton, F. (2005, July). Far voxels: A multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms. In *ACM SIGGRAPH 2005 Papers* (pp. 878–885). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1186822.1073277

Goldfeather, J., Hultquist, J. P. M., & Fuchs, H. (1986, August). Fast constructive-solid geometry display in the pixel-powers graphics system. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (pp. 107–116). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/15922.15898

Gross, M., & Pfister, H. (2007). *Point-based graphics*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Groueix, T., Fisher, M., Kim, V. G., Russell, B. C., & Aubry, M. (2018). A Papier-Mâché Approach to Learning 3D Surface Generation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 216–224).

Gruber, A., Fratarcangeli, M., Zoss, G., Cattaneo, R., Beeler, T., Gross, M., & Bradley, D. (2020, August). Interactive Sculpting of Digital Faces Using an Anatomical Modeling Paradigm. *Computer Graphics Forum*, *39*(5), 93–102. doi: 10.1111/cgf.14071

Grünbaum, B., & Shephard, G. C. (1987). *Tilings and patterns* (First ed.). New York: W. H. Freeman and Company.

Guérin, É., Digne, J., Galin, É., Peytavie, A., Wolf, C., Benes, B., & Martinez, B. (2017, November). Interactive example-based terrain authoring with conditional generative adversarial networks. *ACM Transactions on Graphics*, *36*(6), 228:1–228:13. doi: 10.1145/3130800.3130804

Guérin, E., Peytavie, A., Masnou, S., Digne, J., Sauvage, B., Gain, J., & Galin, E. (2022). Gradient Terrain Authoring. *Computer Graphics Forum*, *41*(2).

Gumin, M. (2016). *Wave Function Collapse.*

Guy, E., Thiery, J.-M., & Boubekeur, T. (2014, May). SimSelect: Similarity-based selection for 3D surfaces. *Computer Graphics Forum*, *33*(2), 165–173. doi: 10.1111/cgf.12306

Haeberli, P. E. (1988, June). ConMan: A visual programming language for interactive graphics. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques* (pp. 103–111). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/54852.378494

Haegler, S., Wonka, P., Arisona, S. M., Van Gool, L., & Müller, P. (2010). Grammar-based encoding of facades. *Computer Graphics Forum*, *29*(4), 1479–1487. doi: 10.1111/j.1467-8659.2010.01745.x

Hahn, F., Martin, S., Thomaszewski, B., Sumner, R., Coros, S., & Gross, M. (2012, July). Rig-space physics. *ACM Trans. Graph.*, *31*(4). doi: 10.1145/2185520.2185568

Hart, J. C. (1996, December). Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, *12*(10), 527–545. doi: 10.1007/s003710050084

Hastings, W. K. (1970, April). Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, *57*(1), 97–109. doi: 10.1093/biomet/57.1.97

Havemann, S. (2005). *Generative mesh modeling* (Doctoral dissertation). doi: 10.24355/dbbs.084-200603150100-7

Hecher, M., Guerrero, P., Wonka, P., & Wimmer, M. (2018, August). How Do Users Map Points Between Dissimilar Shapes? *IEEE Transactions on Visualization and Computer Graphics*, *24*(8), 2327–2338. doi: 10.1109/TVCG.2017.2730877

Heitz, E., Dupuy, J., Crassin, C., & Dachsbacher, C. (2015). The SGGX microflake distribution. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*, *34*(4), 48:1–48:11.

Hempel, B., & Chugh, R. (2016, October). Semi-Automated SVG Programming via Direct Manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (pp. 379–390). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2984511.2984575

Hempel, B., Lubin, J., & Chugh, R. (2019, October). Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (pp. 281–292). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3332165.3347925

Hilaga, M., Shinagawa, Y., Kohmura, T., & Kunii, T. L. (2001). Topology matching

for fully automatic similarity estimation of 3D shapes. In *Proceedings of the 28th annual conference on computer graphics and interactive techniques* (pp. 203–212). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/383259.383282

Hils, D. D. (1992, March). Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, *3*(1), 69–101. doi: 10.1016/1045-926X(92)90034-J

Hirschberg, D. S. (1975, June). A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, *18*(6), 341–343. doi: 10.1145/360825.360861

Holden, D., Saito, J., & Komura, T. (2015, August). Learning an inverse rig mapping for character animation. In *Proceedings of the 14th ACM SIGGRAPH / Eurographics Symposium on Computer Animation* (pp. 165–173). Los Angeles California: ACM. doi: 10.1145/2786784.2786788

Hoppe, H. (1996). Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (pp. 99–108).

Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., & Stuetzle, W. (1993). Mesh optimization. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques* (pp. 19–26).

Hu, Y., Dorsey, J., & Rushmeier, H. (2019, November). A novel framework for inverse procedural texture modeling. *ACM Transactions on Graphics*, *38*(6), 186:1–186:14. doi: 10.1145/3355089.3356516

Hu, Y., He, C., Deschaintre, V., Dorsey, J., & Rushmeier, H. (2022, January). An Inverse Procedural Modeling Pipeline for SVBRDF Maps. *ACM Transactions on Graphics*, *41*(2), 18:1–18:17. doi: 10.1145/3502431

Hutchins, E. L., Hollan, J. D., & Norman, D. A. (1985, December). Direct Manipulation Interfaces. *Human–Computer Interaction*, *1*(4), 311–338. doi: 10.1207/s15327051hci0104_2

Igarashi, T., Moscovich, T., & Hughes, J. F. (2005, July). As-rigid-as-possible shape manipulation. *ACM Trans. Graph.*, *24*(3), 1134–1141. doi: 10.1145/1073204.1073323

Jacobson, A., Baran, I., Popović, J., & Sorkine, O. (2011, July). Bounded biharmonic weights for real-time deformation. In *ACM SIGGRAPH 2011 papers* (pp. 1–8). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1964921.1964973

Jeschke, S., Mantler, S., & Wimmer, M. (2007, June). Interactive smooth and curved shell mapping. In *Proceedings of the 18th Eurographics conference on Rendering Techniques* (pp. 351–360). Goslar, DEU: Eurographics Association.

Johnson, J., Hariharan, B., van der Maaten, L., Hoffman, J., Fei-Fei, L., Lawrence Zitnick, C., & Girshick, R. (2017). Inferring and Executing Programs for Visual Reasoning. In *Proceedings of the IEEE International Conference on Computer Vision* (pp. 2989–2998).

Johnston, W. M., Hanna, J. R. P., & Millar, R. J. (2004, March). Advances in dataflow programming languages. *ACM Computing Surveys*, *36*(1), 1–34. doi:

10.1145/1013208.1013209

Jones, R. K., Barton, T., Xu, X., Wang, K., Jiang, E., Guerrero, P., … Ritchie, D. (2020, November). ShapeAssembly: Learning to generate programs for 3D shape structure synthesis. *ACM Trans. Graph.*, *39*(6). doi: 10.1145/3414685.3417812

Jones, R. K., Charatan, D., Guerrero, P., Mitra, N. J., & Ritchie, D. (2021, July). ShapeMOD: Macro operation discovery for 3D shape programs. *ACM Transactions on Graphics*, *40*(4), 153:1–153:16. doi: 10.1145/3450626.3459821

Ju, T., Schaefer, S., & Warren, J. (2005). Mean value coordinates for closed triangular meshes. *ACM Trans. Graph.*, *24*(3), 561–566.

Kaiser, A., Ybanez Zepeda, J. A., & Boubekeur, T. (2019). A Survey of Simple Geometric Primitives Detection Methods for Captured 3D Data. *Computer Graphics Forum*, *38*(1), 167–196. doi: 10.1111/cgf.13451

Kalojanov, J., Wand, M., & Slusallek, P. (2016). Building Construction Sets by Tiling Grammar Simplification. *Computer Graphics Forum*, *35*(2), 13–25. doi: 10.1111/cgf.12807

Kämpe, V., Sintorn, E., & Assarsson, U. (2013, July). High resolution sparse voxel DAGs. *ACM Trans. Graph.*, *32*(4), 101:1–101:13. doi: 10.1145/2461912.2462024

Karabela, T. (2020). *MfxVTK: An OpenMfx plug-in based on the visualization toolkit (VTK).* https://github.com/tkarabela/MfxVTK.

Keinert, B., Innmann, M., Sänger, M., & Stamminger, M. (2015). Spherical fibonacci mapping. *ACM Transactions on Graphics (TOG)*, *34*(6), 193.

Kelley, A. D., Malin, M. C., & Nielson, G. M. (1988, June). Terrain simulation using a model of stream erosion. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques* (pp. 263–268). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/54852.378519

Kelly, G., & McCabe, H. (2006). A survey of procedural techniques for city generation. *The ITB Journal*, *7*(2), 5.

Kelly, T., Wonka, P., & Mueller, P. (2015, May). Interactive Dimensioning of Parametric Models. *Computer Graphics Forum*, *34*(2), 117–129. doi: 10.1111/cgf.12546

Kilian, M., Mitra, N. J., & Pottmann, H. (2007, July). Geometric modeling in shape space. In *ACM SIGGRAPH 2007 papers* (pp. 64–es). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1275808.1276457

Kirsch, F., & Döllner, J. (2004). Rendering techniques for hardware-accelerated image-based CSG. *Journal of WSCG*, *12*(1-3).

Koniaris, C., Cosker, D., Yang, X., & Mitchell, K. (2014, February). Survey of texture mapping techniques for representing and rendering volumetric mesostructure. *Journal of Computer Graphics Techniques*.

Kraevoy, V., & Sheffer, A. (2004, August). Cross-parameterization and compatible remeshing of 3D models. *ACM Trans. Graph.*, *23*(3), 861–869. doi: 10.1145/1015706.1015811

Kripac, J. (1997, February). A mechanism for persistently naming topological entities in history-based parametric solid models. *Computer-Aided Design*,

29(2), 113–122. doi: 10.1016/S0010-4485(96)00040-1

Krispel, U., Schinko, C., & Ullrich, T. (2014). The rules behind – tutorial on generative modeling. *Proceedings of Symposium on Geometry Processing / Graduate School*, *12*, 2:1–2:49.

Krištof, P., Beneš, B., Křivánek, J., & Šťava, O. (2009). Hydraulic Erosion Using Smoothed Particle Hydrodynamics. *Computer Graphics Forum*, *28*(2), 219–228. doi: 10.1111/j.1467-8659.2009.01361.x

Krs, V., Mech, R., Gaillard, M., Carr, N., & Benes, B. (2020). PICO: Procedural Iterative Constrained Optimizer for Geometric Modeling. *IEEE Transactions on Visualization and Computer Graphics*, 1–1. doi: 10.1109/TVCG.2020.2995556

Kurz, C., Wu, X., Wand, M., Thormählen, T., Kohli, P., & Seidel, H.-P. (2014). Symmetry-Aware Template Deformation and Fitting. *Computer Graphics Forum*, *33*(6), 205–219. doi: 10.1111/cgf.12344

Lagae, A., Dumont, O., & Dutre, P. (2005, June). Geometry synthesis by example. In *International Conference on Shape Modeling and Applications 2005 (SMI' 05)* (pp. 174–183). doi: 10.1109/SMI.2005.24

Laidlaw, D. H., Trumbore, W. B., & Hughes, J. F. (1986, August). Constructive solid geometry for polyhedral objects. *SIGGRAPH Comput. Graph.*, *20*(4), 161–170. doi: 10.1145/15886.15904

Landreneau, E., & Schaefer, S. (2010). Scales and Scale-like Structures. *Computer Graphics Forum*, *29*(5), 1653–1660. doi: 10.1111/j.1467-8659.2010.01774.x

Lau, T., Wolfman, S. A., Domingos, P., & Weld, D. S. (2003, October). Programming by Demonstration Using Version Space Algebra. *Machine Learning*, *53*(1), 111–156. doi: 10.1023/A:1025671410623

Lazarus, F., & Verroust, A. (1999, June). Level set diagrams of polyhedral objects. In *Proceedings of the fifth ACM symposium on Solid modeling and applications* (pp. 130–140). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/304012.304025

Lê, E.-T., Sung, M., Ceylan, D., Mech, R., Boubekeur, T., & Mitra, N. J. (2021). CPFN: Cascaded Primitive Fitting Networks for High-Resolution Point Clouds. In *Proceedings of the IEEE/CVF International Conference on Computer Vision* (pp. 7457–7466).

Leaf, J., Wu, R., Schweickart, E., James, D. L., & Marschner, S. (2018, December). Interactive design of periodic yarn-level cloth patterns. *ACM Transactions on Graphics*, *37*(6), 202:1–202:15. doi: 10.1145/3272127.3275105

Leimer, K., Gersthofer, L., Wimmer, M., & Musialski, P. (2017, October). Relation-based parametrization and exploration of shape collections. *Computers & Graphics*, *67*, 127–137. doi: 10.1016/j.cag.2017.07.001

Le Muzic, M., Autin, L., Parulek, J., & Viola, I. (2015). cellVIEW: A tool for illustrative and multi-scale rendering of large biomolecular datasets. In *Eurographics workshop on visual computing for biomedicine* (Vol. 2015, p. 61).

Le Roux, O., Gaildrat, V., & Caubet, R. (2001, July). Using constraint propagation and domain reduction for the generation phase in declarative modeling. In

Proceedings Fifth International Conference on Information Visualisation (pp. 117–123). doi: 10.1109/IV.2001.942047

Lescoat, T., Ovsjanikov, M., Memari, P., Thiery, J.-M., & Boubekeur, T. (2018). A Survey on Data-driven Dictionary-based Methods for 3D Modeling. *Computer Graphics Forum*, *37*(2), 577–601. doi: 10.1111/cgf.13384

Levi, Z., & Gotsman, C. (2015, February). Smooth rotation enhanced as-rigid-as-possible mesh animation. *IEEE Transactions on Visualization and Computer Graphics*, *21*(2), 264–277. doi: 10.1109/TVCG.2014.2359463

Lewis, J. P., & Anjyo, K. (2010, July). Direct Manipulation Blendshapes. *IEEE Computer Graphics and Applications*, *30*(4), 42–50. doi: 10.1109/MCG.2010.41

Li, H., Weise, T., & Pauly, M. (2010, July). Example-based facial rigging. *ACM Trans. Graph.*, *29*(4). doi: 10.1145/1778765.1778769

Li, T., Bolkart, T., Black, M. J., Li, H., & Romero, J. (2017, November). Learning a model of facial shape and expression from 4D scans. *ACM Trans. Graph.*, *36*(6). doi: 10.1145/3130800.3130813

Lieberman, H. (2001). *Your Wish is My Command: Programming by Example* (H. Lieberman, Ed.). San Francisco: Morgan Kaufmann. doi: 10.1016/B978-155860688-3/50001-4

Lieberman, H., Paternò, F., Klann, M., & Wulf, V. (2006). End-User Development: An Emerging Paradigm. In H. Lieberman, F. Paternò, & V. Wulf (Eds.), *End User Development* (pp. 1–8). Dordrecht: Springer Netherlands. doi: 10.1007/1-4020-5386-X_1

Lienhard, S. (2017). *Visualization, adaptation, and transformation of procedural grammars* (Unpublished doctoral dissertation). EPFL.

Lienhard, S., Lau, C., Müller, P., Wonka, P., & Pauly, M. (2017, May). Design Transformations for Rule-based Procedural Modeling. *Computer Graphics Forum*, *36*(2), 39–48. doi: 10.1111/cgf.13105

Lin, A., Lee, G. S., Longson, J., Steele, J., Goldberg, E., & Stefanovic, R. (2015). Achieving real-time playback with production rigs. In *ACM SIGGRAPH 2015 talks* (pp. 11:1–11:1). New York, NY, USA: ACM. doi: 10.1145/2775280.2792519

Lindenmayer, A. (1968, March). Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *Journal of Theoretical Biology*, *18*(3), 280–299. doi: 10.1016/0022-5193(68)90079-9

Lindstrom, P. (2000). Out-of-core simplification of large polygonal models. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (pp. 259–262).

Lintermann, B., & Deussen, O. (1999, January). Interactive modeling of plants. *IEEE Computer Graphics and Applications*, *19*(1), 56–65. doi: 10.1109/38.736469

Lipp, M., Specht, M., Lau, C., Wonka, P., & Müller, P. (2019). Local Editing of Procedural Models. *Computer Graphics Forum*, *38*(2), 13–25. doi: 10.1111/cgf.13615

Liu, H., Vimont, U., Wand, M., Cani, M.-P., Hahmann, S., Rohmer, D., & Mitra, N. J. (2015). Replaceable Substructures for Efficient Part-Based Modeling.

Computer Graphics Forum, *34*(2), 503–513. doi: 10.1111/cgf.12579

Liu, L., Zheng, Y., Tang, D., Yuan, Y., Fan, C., & Zhou, K. (2019, July). NeuroSkinning: Automatic skin binding for production characters with deep graph networks. *ACM Trans. Graph.*, *38*(4). doi: 10.1145/3306346.3322969

Longay, S., Runions, A., Boudon, F., & Prusinkiewicz, P. (2012). TreeSketch: Interactive Procedural Modeling of Trees on a Tablet. In Kara, L.B., Singh, & K. (Eds.), *EUROGRAPHICS Symposium on Sketch-Based Interfaces and Modeling*. Cagliari, Italy.

Loper, M., Mahmood, N., Romero, J., Pons-Moll, G., & Black, M. J. (2015, October). SMPL: A skinned multi-person linear model. *ACM Trans. Graph.*, *34*(6). doi: 10.1145/2816795.2818013

Loubet, G., & Neyret, F. (2017). Hybrid mesh-volume LoDs for all-scale pre-filtering of complex 3D assets. In *Eurographics 2017* (Vol. 36).

Loubet, G., & Neyret, F. (2018, May). A new microflake model with microscopic self-shadowing for accurate volume downsampling. *Computer Graphics Forum*, *37*(2), 111–121. doi: 10.1111/cgf.13346

Maciel, P. W. C., & Shirley, P. (1995). Visual navigation of large environments using textured clusters. In *Proceedings of the 1995 symposium on interactive 3D graphics* (pp. 95–ff.). New York, NY, USA: ACM. doi: 10.1145/199404.199420

Mahmood, N., Ghorbani, N., Troje, N. F., Pons-Moll, G., & Black, M. J. (2019, October). AMASS: Archive of motion capture as surface shapes. In *Proceedings of the IEEE/CVF international conference on computer vision (ICCV)*.

Marchal, L. (2018). *Memory and data aware scheduling* (Habilitation à Diriger Des Recherches). École Normale Supérieure de Lyon.

Marron, A., Weiss, G., & Wiener, G. (2012, October). A decentralized approach for programming interactive applications with JavaScript and blockly. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions* (pp. 59–70). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2414639.2414648

Martinovic, A., & Van Gool, L. (2013, June). Bayesian grammar learning for inverse procedural modeling. In *The IEEE conference on computer vision and pattern recognition (CVPR)*.

Mathur, A., Pirron, M., & Zufferey, D. (2020, September). Interactive Programming for Parametric CAD. *Computer Graphics Forum*, *39*(6), 408–425. doi: 10.1111/cgf.14046

Maung, D., & Crawfis, R. (2015, July). Applying formal picture languages to procedural content generation. In *2015 computer games: AI, animation, mobile, multimedia, educational and serious games (CGAMES)* (pp. 58–64). doi: 10.1109/CGames.2015.7272963

Mayer, M., Kuncak, V., & Chugh, R. (2018, October). Bidirectional evaluation with direct manipulation. *Proceedings of the ACM on Programming Languages*, *2*(OOPSLA), 127:1–127:28. doi: 10.1145/3276497

Mech, R. (1997). *Modeling and simulation of the interaction of plants with the environment using L-systems and their extensions.*

Mei, X., Decaudin, P., & Hu, B.-G. (2007, October). Fast Hydraulic Erosion Simulation and Visualization on GPU. In *15th Pacific Conference on Computer Graphics and Applications (PG'07)* (pp. 47–56). doi: 10.1109/PG.2007.15

Meng, J., Papas, M., Habel, R., Dachsbacher, C., Marschner, S., Gross, M. H., & Jarosz, W. (2015). Multi-scale modeling and rendering of granular materials. *ACM Trans. Graph.*, *34*(4), 49.

Merrell, P. (2007, April). Example-based model synthesis. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games* (pp. 105–112). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1230100.1230119

Merrell, P., & Manocha, D. (2008, December). Continuous model synthesis. In *ACM SIGGRAPH Asia 2008 papers* (pp. 1–7). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1457515.1409111

Merrell, P., Schkufza, E., & Koltun, V. (2010, December). Computer-generated residential building layouts. In *ACM SIGGRAPH Asia 2010 papers* (pp. 1–12). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1866158.1866203

Merrell, P., Schkufza, E., Li, Z., Agrawala, M., & Koltun, V. (2011, July). Interactive furniture layout using interior design guidelines. *ACM Transactions on Graphics*, *30*(4), 87:1–87:10. doi: 10.1145/2010324.1964982

Merry, B., Marais, P., & Gain, J. (2006, October). Animation space: A truly linear framework for character animation. *ACM Trans. Graph.*, *25*(4), 1400–1423. doi: 10.1145/1183287.1183294

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., & Teller, E. (1953, June). Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*, *21*(6), 1087–1092. doi: 10.1063/1.1699114

Meyer, A., & Neyret, F. (2000). Multiscale shaders for the efficient realistic rendering of pine-trees. In *Graphics interface* (pp. 137–144).

Miao, H., Klein, T., Kouřil, D., Mindek, P., Schatz, K., Gröller, M. E., … Viola, I. (2019). Multiscale molecular visualization. *Journal of Molecular Biology*, *431*(6), 1049–1070. doi: 10.1016/j.jmb.2018.09.004

Michel, É. (2019a). *MfxVCG: An OpenMfx plug-in based on VCGlib.* https://github.com/eliemichel/MfxVCG.

Michel, É. (2019b). *OpenMfx for Blender.* https://github.com/eliemichel/OpenMeshEffectForBlender.

Michel, É. (2021, August). OpenMfx: An API for cross-software non-destructible mesh effects. In *ACM SIGGRAPH 2021 Posters* (pp. 1–2). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3450618.3469168

Michel, É., & Boubekeur, T. (2019). Rendu de sable multi-échelle en temps réel. In *Journées Françaises d'Informatique Graphique et de Réalité Virtuelle (JFIGRV).* Marseille, France.

Michel, É., & Boubekeur, T. (2020a). Formes paramétriques différentiables. In *Journées Françaises d'Informatique Graphique (JFIG).* Nancy, France.

Michel, É., & Boubekeur, T. (2020b). Real Time Multiscale Rendering of Dense Dynamic Stackings. *Computer Graphics Forum*, *39*(7), 169–179. doi: 10.1111/cgf.14135

Michel, É., & Boubekeur, T. (2020c, September). Real Time Multi-Scale Sand Rendering. In *Poster*.

Michel, É., & Boubekeur, T. (2021a, July). DAG amendment for inverse control of parametric shapes. *ACM Trans. Graph.*, *40*(4). doi: 10.1145/3450626.3459823

Michel, É., & Boubekeur, T. (2021b). Synthèse par pavage de méso-structure surfacique. In *Journées Françaises d'Informatique Graphique (JFIG)*. Sophia-Antipolis, France.

Michel, E., Emilien, A., & Cani, M.-P. (2015, May). Generation of Folded Terrains from Simple Vector Maps. In *Eurographics 2015 short paper proceedings* (p. 4). The Eurographics Association. doi: 10.2312/egsh.20151019

Miller, C., Arikan, O., & Fussell, D. (2010). Frankenrigs: Building character rigs from multiple sources. In (pp. 31–38). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1730804.1730810

Mitra, N. J., Pauly, M., Wand, M., & Ceylan, D. (2012). Symmetry in 3D geometry: Extraction and applications. In *EUROGRAPHICS state-of-the-art report*. doi: 10.1111/cgf.12010

Mitra, N. J., Wand, M., Zhang, H., Cohen-Or, D., Kim, V., & Huang, Q.-X. (2014, July). Structure-aware shape processing. In *ACM SIGGRAPH 2014 Courses* (pp. 1–21). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2614028.2615401

Moon, J. T., Walter, B., & Marschner, S. R. (2007). Rendering discrete random media using precomputed scattering solutions. In *Proceedings of the 18th Eurographics conference on Rendering Techniques* (pp. 231–242).

Müller, P., Wonka, P., Haegler, S., Ulmer, A., & Van Gool, L. (2006, July). Procedural modeling of buildings. *ACM Trans. Graph.*, *25*(3), 614–623. doi: 10.1145/1141911.1141931

Myers, B. A. (1986, April). Visual programming, programming by example, and program visualization: A taxonomy. *ACM SIGCHI Bulletin*, *17*(4), 59–66. doi: 10.1145/22339.22349

Myers, B. A. (1990, March). Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, *1*(1), 97–123. doi: 10.1016/S1045-926X(05)80036-9

Myers, B. A., Ko, A. J., & Burnett, M. M. (2006, April). Invited research overview: End-user programming. In *CHI '06 Extended Abstracts on Human Factors in Computing Systems* (pp. 75–80). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1125451.1125472

Nandi, C., Wilcox, J. R., Panchekha, P., Blau, T., Grossman, D., & Tatlock, Z. (2018, July). Functional programming for compiling and decompiling computer-aided design. *Proceedings of the ACM on Programming Languages*, *2*(ICFP), 99:1–99:31. doi: 10.1145/3236794

Neyret, F., & Cani, M.-P. (1999, August). Pattern-Based Texturing Revisited. In

26th Annual Conference on Computer Graphics and interactive techniques (SIGGRAPH '99) (p. 235). ACM SIGGRAPH. doi: 10.1145/311535.311561

Nishida, G., Bousseau, A., & Aliaga, D. G. (2018). Procedural Modeling of a Building from a Single Image. *Computer Graphics Forum*, *37*(2), 415–429. doi: 10.1111/cgf.13372

Nishida, G., Garcia-Dorado, I., Aliaga, D. G., Benes, B., & Bousseau, A. (2016, July). Interactive sketching of urban procedural models. *ACM Transactions on Graphics*, *35*(4), 130:1–130:11. doi: 10.1145/2897824.2925951

Olano, M., & Baker, D. (2010). LEAN mapping. In *Proceedings of the 2010 ACM SIGGRAPH symposium on interactive 3D graphics and games* (pp. 181–188).

Osman, A. A. A., Bolkart, T., & Black, M. J. (2020). STAR: A spare trained articulated human body regressor. In *European conference on computer vision (ECCV)*.

Parametric Architecture. (2016). https://parametric-architecture.com/.

Paschalidou, D., Katharopoulos, A., Geiger, A., & Fidler, S. (2021). Neural Parts: Learning Expressive 3D Shape Abstractions With Invertible Neural Networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 3204–3215).

Paschalidou, D., Ulusoy, A. O., & Geiger, A. (2019). Superquadrics Revisited: Learning 3D Shape Parsing Beyond Cuboids. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 10344–10353).

Pascucci, V., Scorzelli, G., Bremer, P.-T., & Mascarenhas, A. (2007). Robust on-line computation of reeb graphs: Simplicity and speed. In *ACM SIGGRAPH 2007 papers* (pp. 58–es). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1275808.1276449

Patow, G. (2011). Procedural Modeling of Suspension Bridges. In *SIACG'2011* (p. 6).

Patow, G. (2012, March). User-friendly graph editing for procedural modeling of buildings. *IEEE Computer Graphics and Applications*, *32*(2), 66–75. doi: 10.1109/MCG.2010.104

Penrose, R. (1974). The role of aesthetics in pure and applied mathematical research. *Bulletin of the Institute of Mathematics and Its Applications*, *10*, 266ff.

Peytavie, A., Dupont, T., Guérin, E., Cortial, Y., Benes, B., Gain, J., & Galin, E. (2019, October). Procedural Riverscapes. *Computer Graphics Forum*.

Policarpo, F., & Oliveira, M. M. (2006, March). Relief mapping of non-height-field surface details. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games* (pp. 55–62). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1111411.1111422

Policarpo, F., Oliveira, M. M., & Comba, J. L. D. (2005). Real-time relief mapping on arbitrary polygonal surfaces. In *Proceedings of the 2005 symposium on interactive 3D graphics and games* (pp. 155–162). New York, NY, USA: ACM. doi: 10.1145/1053427.1053453

Ponjou Tasse, F., Emilien, A., Cani, M.-P., Hahmann, S., & Bernhardt, A. (2014, May). First Person Sketch-based Terrain Editing. In *Graphics Interface 2014* (p. 217). Canadian Information Processing Society Toronto.

Porumbescu, S. D., Budge, B., Feng, L., & Joy, K. I. (2005, July). Shell maps. *ACM Transactions on Graphics*, *24*(3), 626–633. doi: 10.1145/1073204.1073239

Prusinkiewicz, P. (1999). A look at the visual modeling of plants using L-systems. *Agronomie*, *19*(3-4), 211–224.

Prusinkiewicz, P., & Hammel, M. (1993). A Fractal Model of Mountains with Rivers. In *Proceedings of Graphics Interface '93* (pp. 174–180).

Prusinkiewicz, P., Hanan, J., & Měch, R. (2000). An L-System-Based Plant Modeling Language. In M. Nagl, A. Schürr, & M. Münch (Eds.), *Applications of Graph Transformations with Industrial Relevance* (pp. 395–410). Berlin, Heidelberg: Springer. doi: 10.1007/3-540-45104-8_31

Python. (n.d.). *Python's Buffer Protocole.* https://docs.python.org/3/c-api/buffer.html.

Ragan-Kelley, J., Adams, A., Paris, S., Levoy, M., Amarasinghe, S., & Durand, F. (2012, July). Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics*, *31*(4), 32:1–32:12. doi: 10.1145/2185520.2185528

Raunhardt, D., & Boulic, R. (2007, April). Progressive Clamping. In *Proceedings 2007 IEEE International Conference on Robotics and Automation* (pp. 4414–4419). doi: 10.1109/ROBOT.2007.364159

Requicha, A. A. G. a. V. (1977). Constructive Solid Geometry. In *November, 1977. [3] 36 p. : Ill. includes bibliography: p. 31-33.* CUMINCAD.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., … Kafai, Y. (2009, November). Scratch: Programming for all. *Communications of the ACM*, *52*(11), 60–67. doi: 10.1145/1592761.1592779

Ritchie, D., Jobalia, S., & Thomas, A. (2018). Example-based Authoring of Procedural Modeling Programs with Structural and Continuous Variability. *Computer Graphics Forum*, *37*(2), 401–413. doi: 10.1111/cgf.13371

Ritsche, N. (2006, November). Real-time shell space rendering of volumetric geometry. In *Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia* (pp. 265–274). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1174429.1174477

Rossignac, J., & Borrel, P. (1993). Multi-resolution 3D approximations for rendering complex scenes. In *Modeling in computer graphics* (pp. 455–465).

Roth, S. D. (1982). Ray casting for modeling solids. *Computer graphics and image processing*, *18*(2), 109–144.

Rumman, N. A., & Fratarcangeli, M. (2016). State of the art in skinning techniques for articulated deformable characters. In *Proceedings of the 11th joint conference on computer vision, imaging and computer graphics theory and applications: Volume 1: GRAPP* (pp. 200–212). Setubal, PRT: SCITEPRESS - Science and Technology Publications, Lda. doi: 10.5220/0005720101980210

Rusinkiewicz, S., & Levoy, M. (2000). QSplat: A multiresolution point rendering system for large meshes. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (pp. 343–352).

Rutten, D. (2007). *Grasshopper.*

Saab, L., Ramos, O. E., Keith, F., Mansard, N., Souères, P., & Fourquet, J. (2013, April). Dynamic Whole-Body Motion Generation Under Rigid Contacts and Other Unilateral Constraints. *IEEE Transactions on Robotics*, *29*(2), 346–362. doi: 10.1109/TRO.2012.2234351

Sandhu, A., Chen, Z., & McCoy, J. (2019). Enhancing wave function collapse with design-level constraints. In *Proceedings of the 14th international conference on the foundations of digital games.* New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3337722.3337752

Schinko, C., Strobl, M., Ullrich, T., & Fellner, W.-D. (2011). Scripting Technology for Generative Modeling. *International journal on advances in software*, *4*, 308–326.

Schmidt, R., & Wyvill, B. (2005, November). Generalized sweep templates for implicit modeling. In *Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia* (pp. 187–196). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1101389.1101428

Schreiner, J., Asirvatham, A., Praun, E., & Hoppe, H. (2004). Inter-surface mapping. In *ACM SIGGRAPH 2004 papers* (pp. 870–877). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1186562.1015812

Schroeder, W. J., Martin, K., & Lorensen, B. (2006). *The visualization toolkit (4th ed.).* Kitware.

Schulz, A., Shamir, A., Levin, D. I. W., Sitthi-amorn, P., & Matusik, W. (2014, July). Design and fabrication by example. *ACM Transactions on Graphics*, *33*(4), 62:1–62:11. doi: 10.1145/2601097.2601127

Schulz, A., Wang, H., Grinspun, E., Solomon, J., & Matusik, W. (2018, July). Interactive exploration of design trade-offs. *ACM Transactions on Graphics*, *37*(4), 131:1–131:14. doi: 10.1145/3197517.3201385

Scurti, H., & Verbrugge, C. (2018, September). Generating Paths with WFC. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, *14*(1), 271–273.

Sekulic, D. (2004). Efficient occlusion culling. In R. Fernando (Ed.), *GPU gems* (Vol. 1, chap. 29). Addison-Wesley Professional.

Sharma, G., Goyal, R., Liu, D., Kalogerakis, E., & Maji, S. (2018, June). CSGNet: Neural shape parser for constructive solid geometry. In *The IEEE conference on computer vision and pattern recognition (CVPR).*

Sharma, G., Goyal, R., Liu, D., Kalogerakis, E., & Maji, S. (2020). Neural Shape Parsers for Constructive Solid Geometry. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1–1. doi: 10.1109/TPAMI.2020.3044749

Shi, L., Li, B., Hašan, M., Sunkavalli, K., Boubekeur, T., Mech, R., & Matusik, W. (2020, November). Match: Differentiable material graphs for procedural material capture. *ACM Transactions on Graphics*, *39*(6), 196:1–196:15. doi: 10.1145/3414685.3417781

Shin, H., & Igarashi, T. (2007, May). Magic canvas: Interactive design of a 3-D scene

prototype from freehand sketches. In *Proceedings of Graphics Interface 2007* (pp. 63–70). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1268517.1268530

Shneiderman, B. (1981, May). Direct manipulation: A step beyond programming languages (abstract only). In *Proceedings of the Joint Conference on Easier and More Productive Use of Computer Systems. (Part - II): Human Interface and the User Interface - Volume 1981* (p. 143). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/800276.810991

Shugrina, M., Shamir, A., & Matusik, W. (2015, July). Fab forms: Customizable objects for fabrication with validity and geometry caching. *ACM Transactions on Graphics*, *34*(4), 100:1–100:12. doi: 10.1145/2766994

SideFX. (n.d.). *Houdini Engine API.* https://www.sidefx.com/docs/hengine/.

Silva, P. B., Eisemann, E., Bidarra, R., & Coelho, A. (2015, January). Procedural content graphs for urban modeling. *Int. J. Comput. Games Technol.*, *2015*, 10:10–10:10. doi: 10.1155/2015/808904

Silva, P. B., Müller, P., Bidarra, R., & Coelho, A. (2013). Node-based shape grammar representation and editing. In *Proceedings of the workshop on procedural content generation in games (PCG'13)* (pp. 1–8).

Smelik, R., Tutenel, T., de Kraker, K. J., & Bidarra, R. (2010). Integrating procedural generation and manual editing of virtual worlds. In *Proceedings of the 2010 workshop on procedural content generation in games* (pp. 2:1–2:8). New York, NY, USA: ACM. doi: 10.1145/1814256.1814258

Smelik, R. M., De Kraker, K. J., Tutenel, T., Bidarra, R., & Groenewegen, S. A. (2009). A survey of procedural methods for terrain modelling. In *Proceedings of the CASA workshop on 3D advanced media in gaming and simulation (3AMIGAS)* (pp. 25–34).

Smelik, R. M., Tutenel, T., Bidarra, R., & Benes, B. (2014). A survey on procedural modelling for virtual worlds. *Computer Graphics Forum*, *33*(6), 31–50. doi: 10.1111/cgf.12276

Smith, D. C. (1975). *PYGMALION: A Creative Programming Environment* (No. STAN-CS-75-499). Stanford Artificial Intelligence Laboratory.

Solar-Lezama, A. (2008). *Program Synthesis by Sketching* (Unpublished doctoral dissertation). University of California, Berkeley.

Solar-Lezama, A. (2018). *Introduction to Program Synthesis.* https://people.csail.mit.edu/asolar/SynthesisCourse/index.htm.

Song, S. L., Shi, W., & Reed, M. (2020, July). Accurate face rig approximation with deep differential subspace reconstruction. *ACM Transactions on Graphics*, *39*(4), 34:34:1–34:34:12. doi: 10.1145/3386569.3392491

Sorkine, O., & Alexa, M. (2007). As-rigid-as-possible surface modeling. In *Proceedings of the fifth eurographics symposium on geometry processing* (pp. 109–116). Goslar, DEU: Eurographics Association.

Stalberg, O. (2018, April). *Wave function collapse in bad north.* Breda University of Applied Sciences: Everything Procedural Conference on Procedural Content Generation for Games.

Stålberg, O. (2020, June). *Townscaper.* Raw Fury.

Stam, J. (1997). *Aperiodic Texture Mapping.*

Šťava, O., Beneš, B., Brisbin, M., & Křivánek, J. (2008, July). Interactive terrain modeling using hydraulic erosion. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (pp. 201–210). Goslar, DEU: Eurographics Association.

Šťava, O., Beneš, B., Měch, R., Aliaga, D. G., & Krištof, P. (2010). Inverse Procedural Modeling by Automatic Generation of L-systems. *Computer Graphics Forum*, *29*(2), 665–674. doi: 10.1111/j.1467-8659.2009.01636.x

Stava, O., Pirk, S., Kratt, J., Chen, B., Měch, R., Deussen, O., & Benes, B. (2014). Inverse Procedural Modelling of Trees. *Computer Graphics Forum*, *33*(6), 118–131. doi: 10.1111/cgf.12282

Steinberger, M., Kenzel, M., Kainz, B., Wonka, P., & Schmalstieg, D. (2014). On-the-fly generation and rendering of infinite cities on the GPU. *Computer Graphics Forum*, *33*(2), 105–114. doi: 10.1111/cgf.12315

Stiny, G., & Gips, J. (1971). Shape grammars and the generative specification of painting and sculpture. In *IFIP congress (2)* (Vol. 2, pp. 125–135).

Studios, P. A. (2016). *Universal scene description.* https://graphics.pixar.com/usd.

Summers, P. D. (1977, January). A Methodology for LISP Program Construction from Examples. *Journal of the ACM*, *24*(1), 161–175. doi: 10.1145/321992.322002

Sumner, R. W., Zwicker, M., Gotsman, C., & Popović, J. (2005, July). Mesh-based inverse kinematics. *ACM Transactions on Graphics*, *24*(3), 488–495. doi: 10.1145/1073204.1073218

Sutherland, I. E. (1964, May). Sketchpad a Man-Machine Graphical Communication System. *SIMULATION*, *2*(5), R-3. doi: 10.1177/003754976400200514

Szirmay-Kalos, L., & Umenhoffer, T. (2008). Displacement Mapping on the GPU — State of the Art. *Computer Graphics Forum*, *27*(6), 1567–1592. doi: 10.1111/j.1467-8659.2007.01108.x

Takayama, K., Schmidt, R., Singh, K., Igarashi, T., Boubekeur, T., & Sorkine, O. (2011). GeoBrush: Interactive Mesh Geometry Cloning. *Computer Graphics Forum*, *30*(2), 613–622. doi: 10.1111/j.1467-8659.2011.01883.x

Talton, J. O., Gibson, D., Yang, L., Hanrahan, P., & Koltun, V. (2009, December). Exploratory modeling with collaborative design spaces. *ACM Transactions on Graphics*, *28*(5), 1–10. doi: 10.1145/1618452.1618513

Talton, J. O., Lou, Y., Lesser, S., Duke, J., Měch, R., & Koltun, V. (2011, April). Metropolis procedural modeling. *ACM Trans. Graph.*, *30*(2), 11:1–11:14. doi: 10.1145/1944846.1944851

Tan, P., Lin, S., Quan, L., Guo, B., & Shum, H. (2008). Filtering and rendering of resolution-dependent reflectance models. *IEEE Transactions on Visualization and Computer Graphics*, *14*(2), 412–425.

Tarini, M. (2017, September). *Lecture notes in game dev.* Istituto di Scienza e Tecnologie dell'Informazione.

Teboul, O., Kokkinos, I., Simon, L., Koutsourakis, P., & Paragios, N. (2011). Shape grammar parsing via reinforcement learning. In *CVPR 2011* (pp. 2273–2280).

Tierny, J., Vandeborre, J.-P., & Daoudi, M. (2006, October). 3D mesh skeleton extraction using topological and geometrical analyses. In *14th pacific conference on computer graphics and applications (pacific graphics 2006)* (p. s1poster). Tapei, Taiwan.

Todt, S., Rezk-Salama, C., Kolb, A., & Kuhnert, K. (2007). *Fast (spherical) light field rendering with per-pixel depth* (Tech. Rep.). Germany: University of Siegen.

Togelius, J., Yannakakis, G. N., Stanley, K. O., & Browne, C. (2011, September). Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, *3*(3), 172–186. doi: 10.1109/TCIAIG.2011.2148116

Tsao, Y.-F., & Fu, K.-S. (1984). Stochastic skeleton modeling of objects. *Computer Vision, Graphics, and Image Processing*, *25*(3), 348–370. doi: 10.1016/0734-189X(84)90200-7

Tulsiani, S., Su, H., Guibas, L. J., Efros, A. A., & Malik, J. (2017). Learning Shape Abstractions by Assembling Volumetric Primitives. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 2635–2643).

Umetani, N. (2017). Exploring generative 3D shapes using autoencoder networks. In *SIGGRAPH asia 2017 technical briefs*. New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3145749.3145758

Vanegas, C. A., Garcia-Dorado, I., Aliaga, D. G., Benes, B., & Waddell, P. (2012, November). Inverse design of urban procedural models. *ACM Transactions on Graphics*, *31*(6), 168:1–168:11. doi: 10.1145/2366145.2366187

van Kaick, O., Zhang, H., Hamarneh, G., & Cohen-Or, D. (2011). A Survey on Shape Correspondence. *Computer Graphics Forum*, *30*(6), 1681–1707. doi: 10.1111/j.1467-8659.2011.01884.x

Vesdapunt, N., Rundle, M., Wu, H., & Wang, B. (2020). JNR: Joint-based neural rig representation for compact 3D face modeling. *ECCV*, 389–405. doi: 10.1007/978-3-030-58523-5_23

Wade, L., & Parent, R. E. (2000, May). Fast, fully-automated generation of control skeletons for use in animation. In *Computer animation* (p. 164). Los Alamitos, CA, USA: IEEE Computer Society. doi: 10.1109/CA.2000.889075

Wampler, K. (2016, November). Fast and reliable example-based mesh IK for stylized deformations. *ACM Trans. Graph.*, *35*(6). doi: 10.1145/2980179.2982433

Wang, C., Cheung, A., & Bodik, R. (2017, June). Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 452–466). Barcelona Spain: ACM. doi: 10.1145/3062341.3062365

Wang, H. (1961). Proving Theorems by Pattern Recognition — II. *Bell System Technical Journal*, *40*(1), 1–41. doi: 10.1002/j.1538-7305.1961.tb03975.x

Wang, L., Wang, X., Tong, X., Lin, S., Hu, S., Guo, B., & Shum, H.-Y. (2003, July). View-dependent displacement mapping. *ACM Transactions on Graphics*, *22*(3), 334–339. doi: 10.1145/882262.882272

Wang, X., Tong, X., Lin, S., Hu, S., Guo, B., & Shum, H.-Y. (2004). *Generalized Displacement Maps*. The Eurographics Association. doi: 10.2312/EGWR/EGSR04/227-233

Wang, X., Zhang, L., Xie, T., Xiong, Y., & Mei, H. (2012, November). Automating presentation changes in dynamic web applications via collaborative hybrid analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (pp. 1–11). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2393596.2393614

Watt, M., Coumans, E., ElKoura, G., Henderson, R., Kraemer, M., Lait, J., & Reinders, J. (2014). *Multithreading for visual effects*. CRC Press.

Watt, M., Cutler, L. D., Powell, A., Duncan, B., Hutchinson, M., & Ochs, K. (2012). LibEE: A multithreaded dependency graph for character animation. In *Proceedings of the digital production symposium* (pp. 59–66). New York, NY, USA: ACM. doi: 10.1145/2370919.2370930

Whiting, E., Ochsendorf, J., & Durand, F. (2009, December). Procedural modeling of structurally-sound masonry buildings. In *ACM SIGGRAPH Asia 2009 papers* (pp. 1–9). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1661412.1618458

Winston, P. H. (1970, September). Learning Structural Descriptions from Examples. *AI Technical Reports*(231).

Wonka, P., Wimmer, M., & Schmalstieg, D. (2000, June). Visibility preprocessing with occluder fusion for urban walkthroughs. In B. Péroche & H. Rushmeier (Eds.), *Rendering techniques 2000 (proceedings eurographics workshop on rendering)* (pp. 71–82). held in Brno, Czech Republic, June 26-28, 2000: Springer-Verlag Wien New York.

Wonka, P., Wimmer, M., Sillion, F., & Ribarsky, W. (2003, July). Instant architecture. *ACM Transactions on Graphics*, *22*(3), 669–677. doi: 10.1145/882262.882324

Wu, F., Yan, D.-M., Dong, W., Zhang, X., & Wonka, P. (2014, July). Inverse procedural modeling of facade layouts. *ACM Transactions on Graphics*, *33*(4), 121:1–121:10. doi: 10.1145/2601097.2601162

Wu, K., & Yuksel, C. (2017, February). Real-time fiber-level cloth rendering. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (pp. 1–8). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3023368.3023372

Wu, X., Wand, M., Hildebrandt, K., Kohli, P., & Seidel, H.-P. (2014). Real-Time Symmetry-Preserving Deformation. *Computer Graphics Forum*, *33*(7), 229–238. doi: 10.1111/cgf.12491

Xu, C., Wang, R., Zhao, S., & Bao, H. (2017). Real-time linear BRDF MIP-Mapping. In *Eurographics symposium on rendering*.

Xu, Z., Zhou, Y., Kalogerakis, E., & Singh, K. (2019). Predicting animation skeletons for 3D articulated models via volumetric nets. In *2019 international conference on 3D vision (3DV)* (pp. 298–307). doi: 10.1109/3DV.2019.00041

Yang, Y., Barnes, C., & Finkelstein, A. (2022). Learning from Shader Program Traces. *Computer Graphics Forum*, 16.

Yang, Y., Inala, J. P., Bastani, O., Pu, Y., Solar-Lezama, A., & Rinard, M. (2021). Program Synthesis Guided Reinforcement Learning for Partially Observed Environments. In *Advances in Neural Information Processing Systems* (Vol. 34, pp. 29669–29683). Curran Associates, Inc.

Yuksel, C., Kaldor, J. M., James, D. L., & Marschner, S. (2012, July). Stitch meshes for modeling knitted clothing with yarn-level detail. *ACM Trans. Graph.*, *31*(4), 37:1–37:12. doi: 10.1145/2185520.2185533

Zanni, C., Claux, F., & Lefebvre, S. (2018, May). HCSG: Hashing for real-time CSG modeling. In *Proceedings of the ACM SIGGRAPH symposium on interactive 3D graphics and games.* Montreal, Canada. doi: 10.1145/3203198

Zhang, J., Nie, X., & Feng, J. (2020). Inference Stage Optimization for Cross-scenario 3D Human Pose Estimation. *Advances in Neural Information Processing Systems*, *33*, 2408–2419.

Zhao, S., Luan, F., & Bala, K. (2016, July). Fitting procedural yarn models for realistic cloth rendering. *ACM Transactions on Graphics*, *35*(4), 51:1–51:11. doi: 10.1145/2897824.2925932

Zhao, S., Wu, L., Durand, F., & Ramamoorthi, R. (2016). Downsampling scattering parameters for rendering anisotropic media. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, *35*(6), 166:1–166:11.

Zhou, H., Sun, J., Turk, G., & Rehg, J. M. (2007, July). Terrain Synthesis from Digital Elevation Models. *IEEE Transactions on Visualization and Computer Graphics*, *13*(4), 834–848. doi: 10.1109/TVCG.2007.1027

Zhou, K., Huang, X., Wang, X., Tong, Y., Desbrun, M., Guo, B., & Shum, H.-Y. (2006, July). Mesh quilting for geometric texture synthesis. In *ACM SIGGRAPH 2006 Papers* (pp. 690–697). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1179352.1141942

Appendices

A Extra DAG Amendments

Figures VI.1 to VI.4 show extra results for the DAG Amendment of Section III.3.3.

B OpenMfx: Standardization of shape operators

Non-destructive operations are widely used in mesh-based 3D modeling to add procedural effects on top a coarse geometry while allowing it to remain editable. Common such operations include surface subdivision, beveling, repetition, boolean operations. Combined together as a stack or even as a direct acyclic graph, they are a very powerful tool to build parametric assets. Although this mechanism is present in many different 3D modeling suites (Maya, Houdini, Blender, Cinema4D, 3ds Max, etc.), it is not easy to share a parametric asset across them. Indeed, they do not all implement the exact same set of operations, and even when they do, there might be slight discrepancies in their behavior. We designed a plug-in API that enables one to have all supporting 3D modeling suites share the same implementation of a given non-destructive effect. It thus becomes possible to share scenes featuring non-destructive effect without having to destructively bake them.

B.1 Technical approach

We build on top of OpenFX (Association, 2006), an industry standard plug-in API that has been developed by Foundry while facing a very similar problem in the field of compositing, which is nothing else than non-destructive image editing. OpenFX has been designed from the ground up with modularity in mind so we

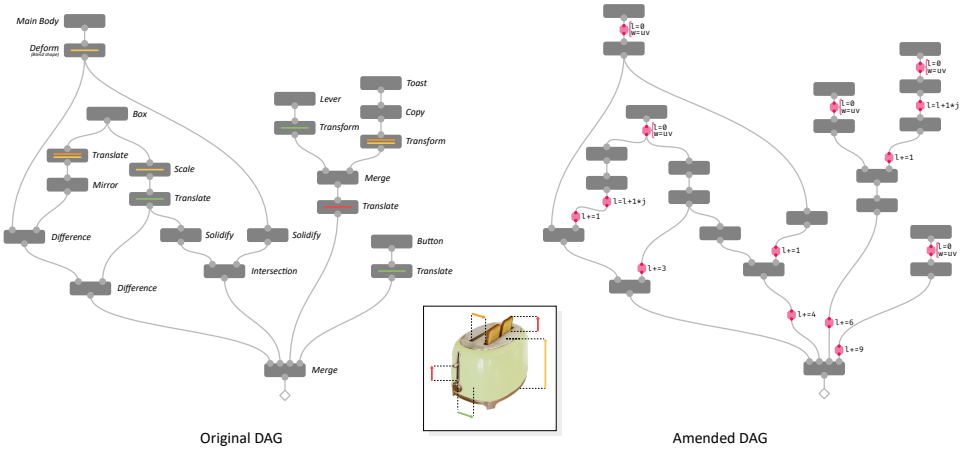


Figure VI.1: *Original DAG (left) and our automatic DAG Amendment (right) for the example of Figure III.8. Colored lines show the hyper-parameters influencing an individual operation.*

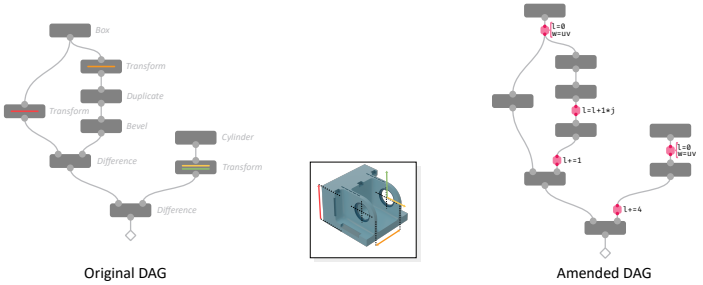


Figure VI.2: *Original DAG (left) and our automatic DAG Amendment (right) for the example (a) of Figure III.24.*

were able to fully reuse its core, including the base plug-in mechanism and the API

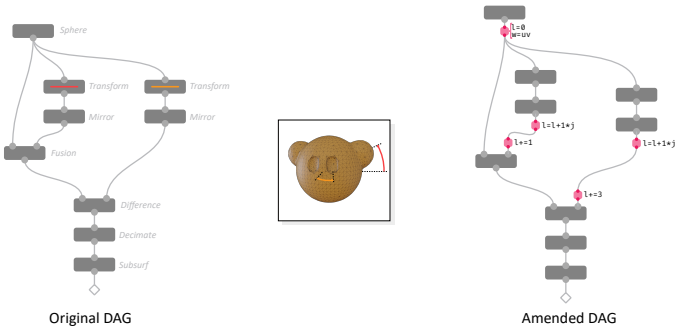


Figure VI.3: *Original DAG (left) and our automatic DAG Amendment (right) for the example (b) of Figure III.24.*

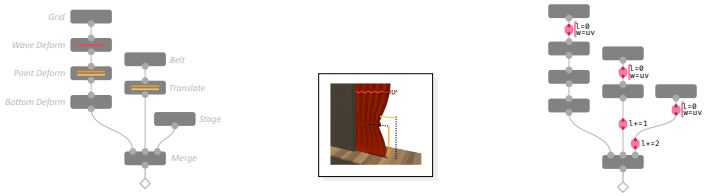


Figure VI.4: *Original DAG (left) and our automatic DAG Amendment (right) for the example (d) of Figure III.24.*

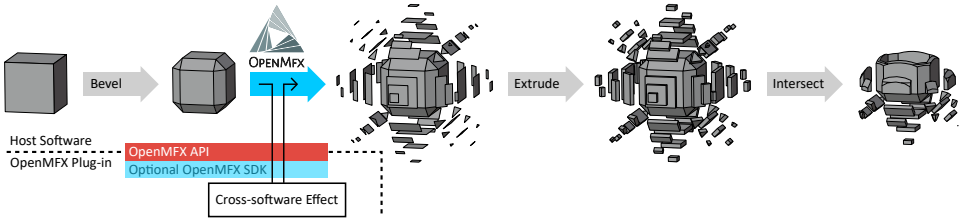


Figure VI.5: *An example of sequence of non-destructive effects transforming a simple editable base mesh (e.g., a cube) into a more complex asset. Operations like bevel, extrusion or boolean intersection are available in most of common mesh-based 3D modeling suites, but the effect applied at the second step is less common. By implementing it as an OpenMfx plug-in, it is available in all supporting host software and thus the asset is interexchangable in its non-destructive form.*

for setting effects' parameters. Our mesh effect API is introduced as an extension next to its image effect API.

OpenMfx' mesh representation is based on a list of *points*, a list of n-gon *faces* and a list of face *corners* sorted by face and referencing point indices. At minimum, points have a position, faces have a vertex count and corners have a point index, but OpenMfx supports for any extra attributes attached to either one of these three entities. This is inspired by the flexibility that made the success of Houdini and its engine (SideFX, n.d.).

We introduced as little overhead as possible, and in particular limit the need for memory duplication. Data buffers for each point/corner/face attributes is given by a pointer, a type (short, int, float), a component count (for vectors) and a stride, which allows in most of the times to use the host's internal memory as is. The API also features mechanism to advertise some extra attributes as either required or useful but optional, so that only what is needed is provided to the OpenMfx effect.

Currently, the API has been developed and documented, one host is supported (a branch of Blender (É. Michel, 2019b)) and another one (in Unity) is at the stage of proof of concept. Several effect packages are available, like *MfxVCG* (É. Michel, 2019a) providing effects from *MeshLab*'s *VCGlib* (et al. Cignoni, 2008) or *MfxVTK*

(Karabela, 2020) providing effects from the *Visualization Toolkit* (Schroeder et al., 2006), and developing new plug-ins is made easy to thanks to an optional C++ helper library (the OpenMfx SDK). Similarly to OpenFX supporting hosts that are either layer-based or node-based, OpenMfx can fit into both stack-based (modifiers) and node-based non-destructive modeling tools.

B.2 Design Choices

The overall plug-in architecture, the notion of effect, of parameter, property etc. could be reused from OpenFX. In the end, most of the decisions we have made were related to the representation of meshes that transits through the API.

This representation aims at supporting a wide variety of mesh topology, including n-gons of arbitrary point count, unconnected points (for point clouds), loose edges (for wireframe meshes) and mixes of all of these. It aims at a minimal enough memory footprint, meaning that the representation should be non-redundant and also that it should be able to point to wherever the data already is in the host's memory rather than copying it, if possible. And it aims at a simple design, to avoid dealing with multiple particular cases.

It is not uncommon for modeling tools to expose two different APIs to handle 3D meshes. For instance 3ds Max has both `Mesh` and `MNMesh`, Blender has `Mesh` and `BMesh`. In both cases, the second one is more flexible and eases arbitrary traversal, but at the cost of some overhead. Our representation is closer to the first ones, the lower level ones. It is then up to the plugin to build a different representation, if needed only (MfxVTK does this for instance).

Attributes An OpenMfx mesh is then simply a list of attribute buffers. A given attribute is relative to either points, faces or face corners and contains between 1 and 4 values of a given type (short, int or float) for each point/face/corner. For instance the vertex positions are given by a 3-component float point attribute. The connectivity information makes no exception, it is given by attributes, namely an integer face attribute telling for each face its number of corners and an integer corner attribute telling for each corner the index of the point it refers to. These are equivalent to what USD's `UsdGeomMesh` class calls resp. `faceVertexIndices` and `faceVertexCounts`, and is also close to Blender's and Houdini's representations[1].

Note that we decided not provide edge attributes. Edges that belong to no face, called "lose edges" are listed as two-point faces, other edges are omitted because implicitly defined by faces. Since edges are shared across faces, the price in clarity to include edge attributes was too important compared to the fact that many existing API don't support them anyways. When really needed, they may be

[1] Houdini calls "vertex" what we call "corner", but "vertex" used by Blender to mean what we and Houdini call "point", we settled on the less ambiguous term "corner".

stored in corners, at the price of duplication, or by explicitly listing all edges as two-point faces.

Attribute buffers are of two kinds. Some own their data, which is freed when the mesh is released. Others are non-owner attributes, meaning that they point to an existing memory location, that is assumed to remain valid throughout the execution of the effect. The memory layout is described by a flexible *buffer protocol*, loosely inspired by Python's (Python, n.d.), such that the host can use non-owner attributes as much as possible to feed mesh attributes to the effect. Only attributes whose layout on host side cannot fit the buffer protocol need to be copied. Non-owner attributes can also be used in outputs, for instance to forward unchanged input attributes without copying them at all.

On-demand data To alleviate further the need for memory transfers, an effect must explicitly request the attributes it needs, when describing its inputs. A requested attribute can be deemed mandatory or not, and is also assigned a *semantic* flag hinting about the meaning of the attribute (color, normal, texture coordinate, weight) and that the host might use to suggest the user which host-side attribute to feed as the requested one. This also brings flexibility since the host then holds a mapping between the actual attribute data it stores for a mesh and what the effect requested.

Another information that is provided only on demand is the world to local transform matrix associated to the mesh. All coordinates are given in local space, and this matrix is not available unless requested, such that the host's dependency graph can more finely avoid useless executions or dependency loops.

SDK We were pursuing competing goals by designing a low overhead API but also at the same time looking for ease of use by developers. This is why on top of the low-level portable C API we provide an optional helper library, written in C++ to provide higher level abstraction. While the low-level API hides no implicit behavior, the C++ SDK.

Specific Scenarios While aiming at flexibility, it was also important to ensure that some more specific yet very common cases are efficiently handled. Often the number of points per face is fixed. It might be to 3 (triangle only meshes), 4 (quad only), but also 2 (wireframe only). In such a case, a flag is used to avoid allocating a face point count buffer that would be uniform.

Attribute forwarding is already a good way to avoid unneeded copies of memory, but for the host's internal it might be useful to know even before running the effect that it will not change the connectivity of the mesh (e.g. a displacement or smoothing effect). So called "deform only" effects can advertise this fact ahead of execution (at describe time) so that the host may handle them differently.

Since the way we handle loose edges was a less obvious decision that may not fit hosts' internal memory layout, and since it is common not to have lose edges in a mesh, we added a flag for the plugin to let the host know when an output mesh has no such edge.

Limitations Besides the aforementioned absence of explicit edge attributes, two features have been set aside compared to Maya's API. First OpenMfx does not allow faces with holes, i.e., faces that would be made of more than one loop of edges. Though technically nothing prevents one from having corners pointing to some reserved point index (for instance -1) meaning to start a new loop, we thought that supporting this would abusively complicate the task of plugin writers while use cases are very uncommon. Secondly, there is no indirection between face corners and texture coordinate, so no proper definition of "UV island".

Limitations that we are willing to address include the current impossibility for a host to allow in-place edition of non-owned attributes, and the lack of mechanism to pass-through arbitrary attributes that were not especially requested by the effect but that the host may want to compute for the output anyways (possibly because another effect requested it downstream).

B.3 Future prospects

OpenMfx is ready to be used and the API is getting stable but its ecosystem may still get improvements in order to ease its adoption. The current major limitation is lack of supporting hosts besides Blender and Unity, so we are working on a SDK similar to the one we have built to ease the creation of plug-ins but oriented towards the integration into new hosts. The Unity host is still limited at this stage as we have not defined a proper dependency graph yet.

In order to ensure harmonization among multiple implementations – which is sometimes a problem of OpenFX – we plan on providing an optional validation layer, reporting any inconsistencies in the use of the API. This will come with an overhead but would be used only at development time.

To meet our original goal of bringing the possibility to share scenes that have non-destructive effects and parametric shapes across software suites, we will integrate an USD schema (Studios, 2016) telling which plug-in to instantiate where and with which parameters when exchanging a scene. Effect parameters would be driven by `UsdAttribute` and input mesh would be provided as `UsdRelationship`. The effect could request from their input the extra attributes provided through the `UsdGeomPrimvarsAPI`.

So far we have focused on time independent effects, but the original OpenFX API also supports time-dependent effects such as simulations, so by reusing their industry tested approach we could also support it for 3D meshes even though it may raise questions that are not our priority at the moment.

There is room in the design for augmenting the effects with new actions without breaking compatibility. A promising example would be to add the possibility to query the effect for differential information, e.g. to measure jacobian of the operation, to make it usable in contexts such as machine learning, differentiable rendering or inverse control (É. Michel & Boubekeur, 2021a).

Finally, in practice effects might use other types of representations. We already support polygon soups and point clouds, there is a question whether we should care about other representations such as voxels, heightmaps or implicit surfaces.

C Proof A

In Section IV.3.2.1, we define:

$$Allowed(\bar{T}_1, d_1, d_2) = \bigcup_{t_1(\boldsymbol{\pi}_1)\in\bar{T}_1} \left\{ t_2(\boldsymbol{\pi}_2) \mid i_2 \equiv i_1 \text{ and } \boldsymbol{\pi}_{2d_2} = \boldsymbol{\pi}_{1d_1} \right\} \tag{VI.1}$$

and we need to show that:

$$Allowed(\bar{T}_1, d_1, d_2) = \bigcup_{t_2\in T_\diamond} H_{t_2,d_2}\Big(\bigcup_{\substack{t_1\in T_\diamond \\ \text{st. } i_1\equiv i_2}} V_{t_1,d_1}(\bar{T}_1) \Big)$$

We split the union in Equation VI.1 depending on the value of $i_1 = L_\diamond(t_1, d_1)$, namely the interface type of t_1 in direction d_1:

$$Allowed(\bar{T}_1, d_1, d_2) = \bigcup_{i\in I_\diamond} \bigcup_{\substack{t_1(\boldsymbol{\pi}_1)\in\bar{T}_1 \\ \text{st. } i_1=i}} \left\{ t_2(\boldsymbol{\pi}_2) \mid i_2 \equiv i \text{ and } \boldsymbol{\pi}_{2d_2} = \boldsymbol{\pi}_{1d_1} \right\}$$

We then split the set $\{t_2(\boldsymbol{\pi}_2)\}$ depending on the type t_2 of the tile:

$$Allowed(\bar{T}_1, d_1, d_2) = \bigcup_{i\in I_\diamond} \bigcup_{\substack{t_1(\boldsymbol{\pi}_1)\in\bar{T}_1 \\ \text{st. } i_1=i}} \bigcup_{t_2\in T_\diamond} \left\{ t_2(\boldsymbol{\pi}_2) \mid i_2 \equiv i \text{ and } \boldsymbol{\pi}_{2d_2} = \boldsymbol{\pi}_{1d_1} \right\}$$

The condition that $i_2 \equiv i$ can be moved into the definition of the inner-most union because it does not depend on the hyper-parameters $\boldsymbol{\pi}_2$:

$$Allowed(\bar{T}_1, d_1, d_2) = \bigcup_{i\in I_\diamond} \bigcup_{\substack{t_1(\boldsymbol{\pi}_1)\in\bar{T}_1 \\ \text{st. } i_1=i}} \bigcup_{\substack{t_2\in T_\diamond \\ \text{st. } i_2\equiv i}} \left\{ t_2(\boldsymbol{\pi}_2) \mid \boldsymbol{\pi}_{2d_2} = \boldsymbol{\pi}_{1d_1} \right\}$$

The two inner unions can be swapped:

$$Allowed(\bar{T}_1, d_1, d_2) = \bigcup_{i \in I_\diamond} \bigcup_{\substack{t_2 \in T_\diamond \\ \text{st. } i_2 \equiv i}} \bigcup_{\substack{t_1(\boldsymbol{\pi}_1) \in \bar{T}_1 \\ \text{st. } i_1 = i}} \left\{ t_2(\boldsymbol{\pi}_2) \mid \boldsymbol{\pi}_{2d_2} = \boldsymbol{\pi}_{1d_1} \right\}$$

Since $\equiv$ is an equivalence relation ($i \mapsto j$ st. $j \equiv i$ is a bijection), we can operate a change of variable from i to j in the outermost union:

$$Allowed(\bar{T}_1, d_1, d_2) = \bigcup_{j \in I_\diamond} \bigcup_{\substack{t_2 \in T_\diamond \\ \text{st. } i_2 = j}} \bigcup_{\substack{t_1(\boldsymbol{\pi}_1) \in \bar{T}_1 \\ \text{st. } i_1 \equiv j}} \left\{ t_2(\boldsymbol{\pi}_2) \mid \boldsymbol{\pi}_{2d_2} = \boldsymbol{\pi}_{1d_1} \right\}$$

And we now merge the first two unions:

$$Allowed(\bar{T}_1, d_1, d_2) = \bigcup_{t_2 \in T_\diamond} \bigcup_{\substack{t_1(\boldsymbol{\pi}_1) \in \bar{T}_1 \\ \text{st. } i_1 \equiv i_2}} \left\{ t_2(\boldsymbol{\pi}_2) \mid \boldsymbol{\pi}_{2d_2} = \boldsymbol{\pi}_{1d_1} \right\}$$

We now express the inner set using H_{t_2,d_2}:

$$Allowed(\bar{T}_1, d_1, d_2) = \bigcup_{t_2 \in T_\diamond} \bigcup_{\substack{t_1(\boldsymbol{\pi}_1) \in \bar{T}_1 \\ \text{st. } i_1 \equiv i_2}} H_{t_2,d_2}\left(\{\boldsymbol{\pi}_{1d_1}\}\right)$$

Noting that $H_{t,d}(A \cup B) = H_{t,d}(A) \cup H_{t,d}(B)$, we move the union inside of H:

$$Allowed(\bar{T}_1, d_1, d_2) = \bigcup_{t_2 \in T_\diamond} H_{t_2,d_2}\Big(\bigcup_{\substack{t_1(\boldsymbol{\pi}_1) \in \bar{T}_1 \\ \text{st. } i_1 \equiv i_2}} \{\boldsymbol{\pi}_{1d_1}\} \Big)$$

We now focus on the argument of H, which we note $(*)$:

$$(*) = \bigcup_{\substack{t_1(\boldsymbol{\pi}_1) \in \bar{T}_1 \\ \text{st. } i_1 \equiv i_2}} \{\boldsymbol{\pi}_{1d_1}\}$$

We split the union to iterate first on the tile type and then on the hyper-parameters:

$$(*) = \bigcup_{\substack{t_1 \in T_\diamond \\ \text{st. } i_1 \equiv i_2}} \bigcup_{\substack{\boldsymbol{\pi}_1 \text{ st.} \\ t_1(\boldsymbol{\pi}_1) \in \bar{T}_1}} \{\boldsymbol{\pi}_{1d_1}\}$$

We rephrase the inner union as a set definition:

Macrosurface		Tile set		Mesostructure			Timing (ms)					
				Triangle Count	Memory (KB)		Mapping		Render		Solve	Suggest
Name	Quad Count	Tile Count	Sweep Count		GPU Buffers	Exported Mesh	CPU	GPU	CPU	GPU		
ninja	1 917	4	25	15 627 538	2 254	365 802						
shell	3 088	3	20	24 950 040	2 859	630 056						
lamp	840	5	28	10 907 520	2 230	273 392	0.7	0.2	0.1	3.5	12.8	
lamp	840	4	30	6 502 560	1 647	164 377	0.6	0.2	0.2	3.7	22.3	3.6
lamp	840	5	31	6 422 722	1 764	164 719	0.8	0.3	0.2	3.6	17.8	5
lamp	840	5	11	7 191 120	1 333	179 393	0.5	0.2	0.1	4	43.3	5.2
lamp	840	14	51	6 177 432	3 488	154 565	1.5	0.2	0.2	3.1	39.1	
lamp-sub1	3 360	6	36	48 372 480	4 628	1 212 430	1.1	0.1	0.2	5.8	139	
lamp-sub1	3 360	5	28	48 372 480	4 155	1 212 430	0.9	0.1	0.2	5.9	88	
lamp-sub2	26 880	5	28	202 974 720	12 058	5 087 450	2.8	0.3	0.1	24	1029	
lamp-sub2	26 880	6	36	202 974 720	12 530	5 087 450	2.8	0.3	0.1	22	3090	
shoe	2 984	4	25	24 325 206	2 956	613 703	1	0.1	0.1	5.1	87	
shoe	2 984	5	10	18 823 072	1 832	469 567	0.6	0.1	0.1	5.1	135	2.5
shoe	2 984	5	28	46 981 376	4 046	1 177 563	1	0.1	0.1	6.8	135	
shoe	2 984	9	42	29 758 940	4 179	745 506	1.5	0.1	0.2	5.8	279	3.5
tshirt	7 088	5	10	44 711 104	3 064	1 115 378	0.7	0.1	0.1	6	3212	3.1
tshirt	7 088	4	25	57 899 346	5 655	1 460 653	1.9	0.1	0.2	6.2	1132	
tshirt	7 088	4	25	271 156 236	9 114		1.5	0.1	0.2	30.6	1132	
tshirt	7 088	9	42	69 991 288	6 298	1 752 922	2.2	0.1	0.2	7.9	708	

Table VI.1: *Various metrics for a series of test scenes, including the examples from Figure IV.10. The second to last row is the same setting as the previous one but using a finer resolution when synthesizing the output mesh. It is used to stress test the rendering pipeline.*

$$(*) = \bigcup_{\substack{t_1 \in T_\diamond \\ \text{st. } i_1 \equiv i_2}} \{\boldsymbol{\pi}_{1d_1} \mid t_1(\ldots, \boldsymbol{\pi}_{1d_1}, \ldots) \in \bar{T}_1\}$$

And we know recognize V_{t_1,d_1}:

$$(*) = \bigcup_{\substack{t_1 \in T_\diamond \\ \text{st. } i_1 \equiv i_2}} V_{t_1,d_1}(\bar{T}_1)$$

QED

D Tile Rendering Statistics

Table VI.1 presents additional statistics for the examples of Section V.2.

E Grain Rendering

E.1 Impostor resolution

Section V.3.4.1 asserts that the proportionality factor between the pixel size p of the atlas' maps and the ratio R/L of outer radius w.r.t. distance depends on the camera field of view (fov) and the screen resolution. Full formula is:

$$p = \alpha W \frac{R}{L} \operatorname{cotan}\left(\frac{\text{fov}}{2}\right) \tag{VI.2}$$

Table VI.2: *Maximum and mean angle between two neighbor views for an octahedron of n subdivisions (so N directions)*

n	N	Max (°)	Mean (°)
3	18	90.00	60.00
4	32	66.42	43.29
5	50	54.74	33.57
6	72	42.30	27.39
7	98	35.26	23.12
8	128	32.13	20.01
9	162	29.50	17.63
10	200	26.29	15.75
11	242	23.84	14.24
12	288	21.37	12.99
13	338	19.47	11.94
14	392	18.25	11.05
15	450	17.19	10.28
16	512	16.05	9.62

where W is the image's width in pixels and α is a coefficient a little larger than 1 accounting for the fact that the projection of the bounding sphere of radius R onto the screen is actually an ellipsis, larger than p. Alpha depends on the camera fov and limit distance L: for close grains at high fov, the effect is not negligible, but this is usually out of our scope because we don't use impostors for closer grains. Long story short: we used $\alpha = 1.1$.

E.2 Tables

Table VI.3 evaluates the theoretical formula of Section V.3.4.1 for different resolutions. This table is crossed with measures on octahedron (Table VI.2) to the data used for Figure V.9 of the paper.

Table VI.3: *Angle of the cone of views under which a single precomputed view is valid, as a function of its pixel size p. The higher the resolution, the thiner the cone.*

p	Cone angle (°)
1	120.00
2	73.74
4	39.50
8	20.13
16	10.11
32	5.06
64	2.53
128	1.27
256	0.63
512	0.32

Table VI.4: *Uncompressde weight of an impostor, assuming a G-buffer fragment fits in 64 bit, for different spatial and angular resolutions.*

$n \backslash p$	1	2	4	8	16	32	64	128	256	512
3	144B	576B	2KB	9KB	36KB	144KB	576KB	2MB	9MB	36MB
4	256B	1KB	4KB	16KB	64KB	256KB	1MB	4MB	16MB	64MB
5	400B	1KB	6KB	25KB	100KB	400KB	1MB	6MB	25MB	100MB
6	576B	2KB	9KB	36KB	144KB	576KB	2MB	9MB	36MB	144MB
7	784B	3KB	12KB	49KB	196KB	784KB	3MB	12MB	49MB	196MB
8	1KB	4KB	16KB	64KB	256KB	1MB	4MB	16MB	64MB	256MB
9	1KB	5KB	20KB	81KB	324KB	1MB	5MB	20MB	81MB	324MB
10	1KB	6KB	25KB	100KB	400KB	1MB	6MB	25MB	100MB	400MB
11	1KB	7KB	30KB	121KB	484KB	1MB	7MB	30MB	121MB	484MB
12	2KB	9KB	36KB	144KB	576KB	2MB	9MB	36MB	144MB	576MB
13	2KB	10KB	42KB	169KB	676KB	2MB	10MB	42MB	169MB	676MB
14	3KB	12KB	49KB	196KB	784KB	3MB	12MB	49MB	196MB	784MB
15	3KB	14KB	56KB	225KB	900KB	3MB	14MB	56MB	225MB	900MB
16	4KB	16KB	64KB	256KB	1MB	4MB	16MB	64MB	256MB	1024MB

Table VI.5: *Difference between the mean angle between two precomputed views and the angle of validity of a single direction for different combinations of the impostors' pixel size p and the number of subdivisions of the octahedron n. A value of zero means that the impostor is perfectly valid.*

$n \backslash p$	1	2	4	8	16	32	64	128	256	512
3	**0**	6	31	45	52	56	58	59	59	59
4	**0**	**0**	15	29	36	39	41	42	42	43
5	**0**	**0**	5	19	26	29	31	32	33	33
6	**0**	**0**	**0**	13	20	23	25	26	26	27
7	**0**	**0**	**0**	8	15	19	21	22	22	22
8	**0**	**0**	**0**	5	12	16	18	19	19	19
9	**0**	**0**	**0**	3	10	14	15	16	17	17
10	**0**	**0**	**0**	1	8	12	13	14	15	15
11	**0**	**0**	**0**	**0**	7	10	12	13	13	14
12	**0**	**0**	**0**	**0**	5	9	11	12	12	12
13	**0**	**0**	**0**	**0**	4	8	10	11	11	11
14	**0**	**0**	**0**	**0**	3	7	9	10	10	10
15	**0**	**0**	**0**	**0**	3	6	8	9	9	10
16	**0**	**0**	**0**	**0**	2	6	7	8	9	9

ECOLE DOCTORALE

Titre: Création interactive de formes 3D représentées en tant que programmes

Mots clés: modélisation

Résumé: Malgré la constante amélioration de la technique et du matériel informatique, permettant de manipuler du contenu numérique de plus en plus volumineux, la création de scènes virtuelles 3D reste une tâche complexe ; du fait notamment de la charge cognitive qu'elle impose aux artistes. Nous proposons une série de contributions visant à tirer parti des représentations par programme des formes pour faire du processus de création de scènes numérique 3D une tâche plus artistique et moins technique qu'elle ne l'est.
Nous rendons possible l'utilisation de méthodes de manipulation directe sur la géométrie générée par DAG grâce à un jeu de règles de réécriture automatique et un filtre non linéaire de donnée différentielle. Nous aidons la création de programmes de forme impératifs en transformant des sélection d'éléments géométriques en des requêtes sémantiques, et la création de programmes déclaratifs en proposant un mode d'édition du contenu géométrique de tuiles de Wang centré sur les sections aux interfaces entre tuiles. Nous étendons les moteurs de pavage par tuiles pour prendre en compte des paramètres continus et suggérer automatiquement de nouvelles tuiles à ajouter. Nous intégrons les programmes de forme à la boucle de retour visuel en délégant l'évaluation du contenu des tuiles au système de rendu en temps-réel, et exploitons la sémantique du programme pour dériver un système de niveau de détails par imposteurs visuels.

Title: Interactive Authoring of 3D Shapes Represented as Programs

Keywords: computer graphics, parametric shapes, procedural modeling

Abstract: Although hardware and techniques have become better and better over the years at handling heavy content, digital 3D creation remains fairly complex, partly because the bottleneck also lies in the cognitive load imposed over the designers. We propose a series of contributions aiming at leveraging program-based representations of shapes to make the process of authoring 3D digital scenes more of an artistic act and less of a technical task.
We enable the use of direct manipulation methods on DAG output thanks to automated rewriting rules and a non-linear filtering of differential data. We help the creation of imperative shape programs by turning geometric selection into semantic queries and of declarative programs by proposing an interface-first editing scheme for authoring 3D content in Wang tiles. We extend tiling engines to handle continuous tile parameters and arbitrary slot graphs, and to suggest new tiles to add to the set. We blend shape programs into the visual feedback loop by delegating tile content evaluation to the real-time rendering pipeline or exploiting the program's semantics to drive an impostor-based level-of-details system.

Institut Polytechnique de Paris

91120 Palaiseau, France