Tom Vierjahn

Online Surface Reconstruction From Unorganized Point Clouds With Integrated Texture Mapping

Westfälische Wilhelms-Universität Münster

Fachbereich 10 Mathematik und Informatik

Informatik

Online Surface Reconstruction From Unorganized Point Clouds With Integrated Texture Mapping

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
durch den Fachbereich Mathematik und Informatik
der Westfälischen Wilhelms-Universität Münster

vorgelegt von
Thomas Vierjahn
aus Oberhausen

2015

Typeset with LaTeX2e and the Classic Thesis Style
using the fonts Palatino, URW Classico, and `Inconsolata`.

Dekan:	Professor Dr. Martin Stein
Erster Gutacher:	Professor Dr. Klaus H. Hinrichs
Zweiter Gutacher:	Professor Dr. Achim Clausing
Tag der mündlichen Prüfung:	11.11.2015
Tag der Promotion:	11.11.2015

Ach, rief Humboldt, was sei Wissenschaft denn dann?

Gauß sog an der Pfeife. Ein Mann allein am Schreibtisch. Ein Blatt Papier vor sich, allenfalls noch ein Fernrohr, vor dem Fenster der klare Himmel. Wenn dieser Mann nicht aufgebe, bevor er verstehe. Das sei vielleicht Wissenschaft.

– Daniel Kehlmann, Die Vermessung der Welt

ABSTRACT

Digital representations of the real world are becoming more and more important for different application domains. Individual objects, excavation sites or even complete cities can be digitized with today's technology so that they can, for instance, be preserved as digital cultural heritage, be used as a basis for map creation, or be integrated into virtual environments for mission planning during emergency or disaster response tasks. Robust and efficient surface reconstruction algorithms are inevitable for these applications.

Surface-reconstructing growing neural gas (SGNG) presented in this dissertation constitutes an artificial neural network that takes a set of sample points lying on an object's surface as an input and iteratively constructs a triangle mesh representing the original object's surface. It starts with an initial approximation that gets continuously refined. At any time during execution, SGNG instantly incorporates any modifications of the input data into the reconstruction. If images are available that are registered to the input points, SGNG assigns suitable textures to the constructed triangles. The number of noticeable occlusion artifacts is reduced to a minimum by learning the required visibility information from the input data.

SGNG is based on a family of closely related artificial neural networks. These are presented in detail and illustrated by pseudocode and examples. SGNG is derived according to a careful analysis of these prior approaches. Results of an extensive evaluation indicate that SGNG improves significantly upon its predecessors and that it can compete with other state-of-the-art reconstruction algorithms.

PUBLICATIONS

Some ideas and figures of this dissertation have appeared in the following publications:

Sven Strothoff, Frank Steinicke, Dirk Feldmann, Jan Roters, Klaus Hinrichs, Tom Vierjahn, Markus Dunkel, and Sina Mostafawy. A virtual reality-based simulator for avionic digital service platforms. In *Proc. Joint Virtual Reality Conf. EGVE – EuroVR – VEC / Ind. Track*, 2010

Sven Strothoff, Dirk Feldmann, Frank Steinicke, Tom Vierjahn, and Sina Mostafawy. Interactive generation of virtual environments using MUAVs. In *Proc. IEEE Int. Symp. VR Innovations*, pp. 89 – 96, 2011. doi: 10.1109/ISVRI.2011.5759608

Tom Vierjahn, Guido Lorenz, Sina Mostafawy, and Klaus Hinrichs. Growing cell structures learning a progressive mesh during surface reconstruction – a top-down approach. In *Eurographics 2012 - Short Papers*, 2012. doi: 10.2312/conf/EG2012/short/029-032

Tom Vierjahn, Niklas Henrich, Klaus Hinrichs, and Sina Mostafawy. sgng: Online surface reconstruction based on growing neural gas. Tech. rep., Dept. Computer Science, Univ. Muenster, Germany, 2013

Tom Vierjahn, Jan Roters, Manuel Moser, Klaus Hinrichs, and Sina Mostafawy. Online reconstruction of textured triangle meshes from aerial images. In *1st Eurographics Workshop Urban Data Modelling and Visualisation*, pp. 1–4, 2013. doi: 10.2312/UDMV/UDMV13/001-004

Tom Vierjahn and Klaus Hinrichs. Surface-reconstructing growing neural gas: A method for online construction of textured triangle meshes. *Computers & Graphics*, pp. –, 2015. doi: 10.1016/j.cag.2015.05.016

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS AND ACRONYMS

AABB	Axis-aligned bounding box
BVH	Bounding volume hierarchy
CHL	Competitive Hebbian learning
CPU	Central processing unit
GCS	Growing cell structures
GNG	Growing neural gas
GPU	Graphics processing unit
GSRM	Growing self-reconstruction map
MRF	Markov random field
NG	Neural gas
LOD	Level of detail
PM	Progressive mesh
PSR	Poisson surface reconstruction
RAM	Random access memory
RGB	Red, green, blue color model
RPA	Remotely piloted aircraft
SFM	Structure from motion
SGC	Smart growing cells
SGNG	Surface-reconstructing growing neural gas
SOM	Self-organizing map
SPSR	Screened Poisson surface reconstruction
TRN	Topology representing network
VRIP	Volumetric range image processing

NOTATION

SETS, ETC.:

$\mathcal{B}(\mathbf{p}, r)$	Ball with radius r around $\mathbf{p}$
$\mathcal{D}_{\mathbf{v}_i}$	Voronoi cell of $\mathbf{v}_i$
$\mathcal{D}_{\mathbf{v}_i,\mathbf{v}_j}$	Second-order Voronoi cell of $\mathbf{v}_i, \mathbf{v}_j$
$\mathcal{E}$	Set of edges connecting vertices in a mesh
$\mathcal{F}$	Set of triangles in a mesh
$\mathcal{F}_e\big((\mathbf{v}_i, \mathbf{v}_j)\big)$	Set of triangles adjacent to edge $(\mathbf{v}_i, \mathbf{v}_j)$
$\mathcal{M}$	A triangle mesh
$\mathcal{N}_{\mathbf{v}_i}$	Set of directly connected neighbor vertices of $\mathbf{v}_i$
$(\mathbf{n}_{\mathbf{v}_i})$	Ordered (counter-clockwise) sequence of directly connected neighbor vertices of $\mathbf{v}_i$
$\mathbb{N}$	Set of natural numbers
$\mathcal{P}$	Set of input points
$\mathcal{P}_{\square}$	Sets of exemplary input points (square)
$\mathcal{P}_{\circledcirc}$	Sets of exemplary input points (annulus)
$\mathcal{P}_{\mathbf{v}_i}$	Set of points contained in the Voronoi cell of $\mathbf{v}_i$
$\mathbb{P}(\cdot)$	Power set
$\mathbb{R}$	Set of real numbers
$\mathcal{S}$	Set of *vertex split* operations (Ch. 3)
$\mathcal{S}$	Set of sample points (Ch. 4)
$\mathcal{T}$	Set of images
$\mathcal{T}_{\mathbf{p}_j}$	Set of images that show input point $\mathbf{p}_j$
$\mathcal{V}$	Set of vertices in a mesh
$(\mathcal{V})$	Neighborhood ranking
$\emptyset$	Empty set

POINTS, NEURAL UNITS, VERTICES, EDGES, ETC.:

$\mathbf{p}$	Position* of an input point
$\mathbf{p}_{\xi_t}$	$\mathbf{p}$, randomly selected in iteration t
$\mathbf{u}_{i,j}$	Location of $v_{i,j}$ in the neural network
v_b	Best matching neural unit for a given $\mathbf{p}_{\xi_t}$
$\mathbf{v}_\mathrm{b}$	Position† of vertex assigned to v_b
$\mathbf{v}_\mathrm{c}$	Position† of the second-best matching vertex
v_i	i-th neural unit of the artificial neural network
$\mathbf{v}_i$	Position† of vertex assigned to v_i
$v_{i,j}$	Neural unit in row i, column j of the network
$\mathbf{v}_{i,j}$	Position of vertex assigned to $v_{i,j}$
$(\mathbf{v}_i, \mathbf{v}_j)$	Edge connecting vertices $\mathbf{v}_i, \mathbf{v}_j$

* Used synonymously for the point itself.
† Used synonymously for the neural unit except in the self-organizing map.

POINTS, NEURAL UNITS, VERTICES, EDGES, ETC.: (CONT.)

$\triangle(\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k)$	Triangle with vertices $\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k$
T_i	i-th image in $\mathcal{T}$

OTHER:

$a\big((\mathbf{v}_k, \mathbf{v}_l)\big)$	Age of edge $(\mathbf{v}_k, \mathbf{v}_l)$
$\mathrm{a}_{\max}(t)$	Maximum edge age
$a_{\mathrm{e}}\big((\mathbf{v}_k, \mathbf{v}_l)\big)$	Penalty of edge $(\mathbf{v}_k, \mathbf{v}_l)$
$a_{\triangle}\big(\triangle(\mathbf{v}_k, \mathbf{v}_l, \mathbf{v}_m)\big)$	Penalty of triangle $\triangle(\mathbf{v}_k, \mathbf{v}_l, \mathbf{v}_m)$
$\mathrm{a}_{\max}$	Maximum edge, triangle penalty (SGNG)
α	Factor for decay of vertex activity
α_s	Activity factor on mesh refinement (GNG, GSRM)
β, η	Step sizes for $\mathbf{v}_{\mathrm{b}}$ and its neighbors, respectively
$\mathbf{c}_{\mathbf{p}_j}$	Color assigned to $\mathbf{p}_j$
$\mathbf{c}_{\mathbf{v}_i}$	Color assigned to $\mathbf{v}_i$
e	Reconstruction error
$E_c\big((\mathbf{v}_m, \mathbf{v}_o)\big)$	Regularity error caused by collapsing $(\mathbf{v}_m, \mathbf{v}_o)$
$\mathrm{ec}(\mathbf{v}_o, \mathbf{v}_m)$	*Edge collapse* with source $\mathbf{v}_o$ and target vertex $\mathbf{v}_m$
$\mathrm{es}(\mathbf{v}_m, \mathbf{v}_n)$	*Edge split* with source $\mathbf{v}_m$ and target vertex $\mathbf{v}_n$
ε	A non-existent vertex (PM)
$\epsilon(t)$	Step size (SOM, NG, TRN)
$h_\sigma(t, v_i, v_{\mathrm{b}})$	Lateral influence (SOM)
$h_\lambda\big(t, k(\mathbf{v}_i, \mathbf{p}_{\xi_t}, \mathcal{V})\big)$	Lateral influence (NG, TRN)
$k(\mathbf{v}_i, \mathbf{p}_{\xi_t}, \mathcal{V})$	Neighborhood rank of $\mathbf{v}_i$ (NG, TRN)
$\lambda(t)$	Width of $h_\lambda\big(t, k(\mathbf{v}_i, \mathbf{p}_{\xi_t}, \mathcal{V})\big)$ (NG, TRN)
λ	Iterations for density update (SGNG)
λ_c	Iterations for *edge collapse*
λ_s	Iterations for *vertex split*
$\varphi(\mathbf{v}_i, \mathbf{v}_j)$	Dihedral angle at edge $(\mathbf{v}_i, \mathbf{v}_j)$
q	Reconstruction quality
$\sigma(t)$	Standard deviation of $h_\sigma(t, v_i, v_{\mathrm{b}})$ (SOM)
t	Number of the current iteration
$\mathrm{t}_{\max}$	Maximum number of iterations
$\mathrm{T}_i(\mathbf{w}_{\mathbf{p}_j, \mathrm{T}_i})$	Pixel color at $\mathbf{w}_{\mathbf{p}_j, \mathrm{T}_i}$ in T_i
$\tau(\mathbf{v}_i)$	Activity of $\mathbf{v}_i$
$\vartheta(\mathbf{v}_i)$	Last iteration in which $\mathbf{v}_i$ was active
$\Delta\vartheta_{\max}$	Inactivity threshold
$\Cup$	SGNG union operator
$\mathrm{vs}(\mathbf{v}_m, \mathbf{v}_s, \mathbf{v}_r, \mathbf{v}_o)$	*Vertex split* with source $\mathbf{v}_m$, left $\mathbf{v}_s$, right $\mathbf{v}_r$, and new vertex $\mathbf{v}_o$
$\mathbf{w}_{\mathbf{p}_j, \mathrm{T}_i}$	2D location of $\mathbf{p}_j$ in T_i
$\mathbf{w}_{\mathbf{v}_j, \mathrm{T}_i}$	Learned texture coordinate of $\mathbf{v}_j$ in T_i
$\begin{bmatrix} x & y & z \end{bmatrix}^\top$	3D column vector with components x, y, z
$\lvert\cdot\rvert$	Cardinality of a set
$\lVert\cdot\rVert$	ℓ^2-norm
$:=$	Assignment operator

PREFACE

This dissertation is the result of the work and research which I have conducted from August 2010 to June 2015 at the Visualization and Computer Graphics Research Group at the Department of Computer Science of the University of Muenster and at rmh new media GmbH, Cologne.

First and foremost, I would like to thank my advisor Professor Dr. Klaus Hinrichs for giving me the opportunity to work on surface reconstruction, as well as for his guidance, his encouragement, and his support not only during this work. I am very grateful for his patience, his enthusiasm, and his valuable feedback, no matter where or when we discussed the work.

My thanks go to Professor Dr. Achim Clausing for his valuable suggestions and his willingness to be my co-referee.

Thank you very much, Professor Dr. Sina Mostafawy for many fruitful discussions, new insights, and for your support.

I am thankful for the support and the required leeway offered by rmh new media GmbH, Cologne and the Virtual Reality Group of RWTH Aachen University. I am also indebted to all my colleagues in Muenster, Cologne, and Aachen for the stimulating collaboration, many helpful discussions, and their support.

This work has been conducted partially within the project AVIGLE (avionic digital service platform). This project was part of the High-tech.NRW research program funded by the ministry for Innovation, Science, Research and Technology of the German state Northrhine-Westfalia, and by the European Union.

Last but certainly not least I would like to express my deepest gratitude to my family and friends. Without your unconditional support, encouragement and patience, conducting this work would have been an unattainable goal. — Thank you.

Duisburg, June 2015
Tom Vierjahn

1 INTRODUCTION

Reconstruction of 3D geometry is of major interest for different application domains. Among others, reverse engineering, cultural heritage, urban reconstruction, and surveying are striving for reconstructing detailed digital geometric 3D models of existing real world objects. If images of these objects are available, it is desirable to use them to texture the models.

In this dissertation *surface-reconstructing growing neural gas* (SGNG) is presented, an iterative online algorithm that constructs a triangle mesh from a set of sample points lying on an object's surface. SGNG instantly provides a coarse approximation of the object's shape that gets continuously refined. At any time during reconstruction, additional data can be added to extend the model, and the mesh constructed so far is readily available for visualization. If the sample points are extracted from images, SGNG uses them as textures for the resulting triangle mesh. That way, fine details of the object's surface are visible even on an initially coarse approximation of the object's shape.

1.1 MOTIVATION

Recent developments allow *remotely piloted aircraft* (RPA) to be used in a variety of civil scenarios. If equipped with suitable payload, a swarm of RPA can monitor pollution levels, serve as communication hotspots or relays in order to increase a cellular network's coverage or bandwidth, or it can provide low-altitude aerial images for surveying or to support map creation. Even autonomous package delivery was considered recently by a major electronic commerce company.

Camera-equipped RPA are also especially useful for reconnaissance during emergency or disaster situations or for search and rescue missions: RPA can be used safely in situations that are too dangerous to send human relief-units. RPA can be sent to remote places immediately while a response team is being assembled, and they can increase the area covered by the mission. The aerial images acquired by the RPA are transmitted to the command and control center and are then used for mission planning. In a common approach, the individual images are assembled to create overview visualizations that are combined with existing data, e.g., from geo-databases. However, three dimensional data, i.e., 3D points, can be extracted from the images using stereophotogrammetry if the images are mutually overlapping. From this data a suitable surface reconstruction algorithm can create textured 3D models to serve as a virtual environment of the operation area supporting decision-making and eventually the units in the field [105, 116]. Even the relief-units can be equipped with cameras providing additional images, thus extending the virtual environment. To be useful in such scenarios, surface reconstruction has to provide quick initial approximations of the captured objects that will be continuously refined whenever new data becomes available without discarding the model constructed so far. Thus, especially during prolonged missions, using an incremental online algorithm is inevitable.

Similar scenarios can be conceived for cultural heritage, archaeology, or large scale urban reconstruction where a digital image library is continuously updated either by experts or by crowd work. New images are then automatically used to extend the reconstructed 3D geometry. The model created so far can be displayed using suitable rendering techniques. It provides information about the regions where data is missing, and thus where additional data needs to be acquired. An incremental online algorithm will incorporate the new data by immediately extending the model, thus providing visual feedback whether suitable images have been added. Impressive results for this scenario have been achieved, but by using offline algorithms [2, 48].

Many of the state-of-the-art reconstruction algorithms do not work in the desired way, neither incremental nor online, although data acquisition itself is an incremental process: Whether laser scanning, stereophotogrammetry, or mechanical probing is used, groups of captured 3D points are provided sequentially. However, many reconstruction algorithms require that the input data has been acquired completely before reconstruction can start. Even if they are able to adapt to modified or extended input data they have to recreate at least parts of the model. Many require a regular sampling pattern, e.g., like in depth images, whereas unorganized point clouds as obtained by stereophotogrammetry are a way more general type of input.

In contrast to these approaches, online learning based reconstruction algorithms are generally able to adapt to any modifications of the input data while reconstructing the original surface. They can even start reconstructing as soon as the first input points are available, and they refine their results while more input points are generated. However, existing online learning based approaches require a huge number of input points and rely on post-processing steps for finalizing the mesh representing the reconstructed surface for the points seen so far. If continuous previewing is desired while acquiring data, reconstruction has to be interrupted repeatedly for post-processing. Thus, reconstruction gets costly.

Surface-reconstructing growing neural gas (SGNG) that is presented in this dissertation constitutes an artificial neural network that performs online learning. SGNG has been designed with the aforementioned iterative pipeline in mind, where data acquisition using overlapping images, reconstruction, and visualization are executed in parallel. Therefore, SGNG iteratively constructs a triangle mesh even from unorganized, sparse point clouds representing an object's surface. SGNG reduces the number of erroneously untriangulated holes to a minimum by taking the constructed surface and geometric considerations into account. Thus, at any time during the construction process a triangle mesh is available, e.g., for visualization in order to direct further data acquisition or mission planning. Furthermore, SGNG instantly adapts to any modifications of the input data: New input points may be added, and existing points may be moved or even removed.

Besides approximating the shape and topology of the original surface, SGNG learns how to assign the available images as textures to the triangles of the constructed mesh. By learning the visibility information that is implicitly encoded in the input data, SGNG does not rely on knowing occluding triangles, as previous algorithms did that derived occlusion from the constructed mesh. Thus, SGNG reduces the number of noticeable texturing artifacts to a minimum even if occluding objects are not represented by the input points at all. SGNG improves the learned texture assignment whenever new images become available.

SGNG works incrementally: It operates at any time only locally on a single input point and its neighborhood within the mesh. Thus, it is very well suited even for massively parallel and out-of-core implementations. However, such implementations lie beyond the scope of this dissertation and are therefore left for future work.

1.2 HOW TO READ THIS DISSERTATION

In this section the reader of this dissertation is provided with a little guidance: At first, the usage of margin notes in this work is described. Then, the presentation of pseudocode in a literate programming approach is explained. Finally, the organization of this dissertation is outlined by providing summaries of the individual chapters and by suggesting suitable reading paths.

Notes in the page margin

In this dissertation notes in the page margin are used frequently. They are intended as headlines in order to provide a more fine-grained structure than the one that can be provided by chapters, sections, and subsections. Furthermore, the series of margin notes of a section or a subsection gives a very brief summary of the respective part of the text. They therefore help the the reader navigate this dissertation.

Pseudocode Many of the algorithms that are presented in this dissertation are illustrated by pseudocode. The pseudocode uses mostly mathematical notation and formulae to be generally applicable. The assignment operator $:=$ denotes an immediate update of a set, an attribute, or a variable. In order to avoid that a reader has to switch frequently between text and a separate listing, Knuth's literate programming approach is used [80]: Descriptive text and pseudocode are interleaved. The syntax and the cross referencing scheme used in this dissertation are very similar to the one used by Pharr and Humphreys [104].

The individual code fragments are named with a descriptive title enclosed in angle brackets. Fragments' names are used in the pseudocode to summarize a set of operations. In order to distinguish between usage and definition, the symbol $\equiv$ following a fragment's name indicates its definition, and the corresponding pseudocode is indented. To provide an illustrative example, the fragment ⟨*Count to 10*⟩ presents the pseudocode for a function counting from 1 to 10.

```
⟨Count to 10⟩ ≡
  ⟨Initialization  → p. 6⟩
  while $n < 10$ do
      ⟨Increment n  → p. 6⟩
  end while
```

That way, a reader can understand the basic operation of the code at the desired abstraction level without letting implementation details obfuscate the intended information.

The implementation details of the other fragments—⟨*Initialization*⟩ and ⟨*Increment n*⟩ in the above example—may require more theoretical background or a special context that has to be carefully described. Several paragraphs may be required when describing real production code, but it is straightforward for this example: At first a counter variable n is initialized to zero.

```
⟨Initialization⟩ ≡                                    ← p. 6
  $n := 0$
```

This counter variable is incremented in each iteration of the loop.

```
⟨Increment n⟩ ≡                                       ← p. 6
  $n := n + 1$
```

Cross references are provided to help navigating the code. Whenever a fragment is used, the page number where its definition can be found is printed alongside the fragment's title, preceded by a right arrow (→). The places of a fragment's usage are indicated by printing the page reference(s) preceded by a left arrow (←) at the top right of the fragment's definition. Both arrows thus indicate program flow. Furthermore, notes in the page margin help the reader locate a fragment's definition.

Sometimes, the description of a fragment is split. This is done in order to keep descriptive text and pseudocode close together, or when further functionality is added to an existing fragment. In such cases, the definition of a fragment is extended by a separate code block indicated by the symbol $+\equiv$. If, for instance, the result of the counting has to be printed in the above example, the fragment $\langle$*Count to 10*$\rangle$ is extended by the required output.

```
⟨Count to 10⟩ +≡
  print $n$
```

Organization The following paragraphs outline the scope of this dissertation. They provide a short summary of the content and the contributions of each chapter. Finally, sensible reading paths are suggested.

Chapter 1 The remainder of this chapter gives an overview of existing surface reconstruction techniques. Since SGNG also provides automatic texturing, several existing techniques for texturing a constructed triangle mesh are also presented. The main contribution of each approach is summarized. At the end of this chapter, the scientific publications are outlined that the contributions of this dissertation are based upon.

Chapter 2 Ch. 2 presents a taxonomy of fundamental artificial neural networks that have been applied to surface reconstruction and that SGNG is based upon. The learning algorithm of each neural network is described in detail and illustrated by pseudocode. Examples demonstrate the capabilities and characteristics of the networks. That chapter provides the theoretical background for the development of SGNG. It covers Kohonen's *self-organizing map* [81, 82, 83] and ranges over *neural gas* and the *topology representing network* by Martinetz et al. [91, 92, 93, 94] to Fritzke's *growing cell structures* [49, 51, 53] and *growing neural gas* [52]. The descriptions end with the *growing self-reconstruction map* by do Rêgo et al. [41].

Chapter 3 Ch. 3 evaluates the relationship of Hoppe's *progressive mesh* [62] and Fritzke's *growing cell structures* (GCS) [49, 51, 53]. Both use *edge split* operations to refine a coarser mesh. However, later extensions making GCS suitable for surface reconstruction [70] add *edge collapse* operations to remove superfluous vertices. Therefore, the constructed mesh is by definition no progressive mesh anymore. In that chapter, an improved vertex removal scheme is derived that enables the extended GCS to construct a *progressive mesh*. The resulting data structure is furthermore used as an acceleration structure to identify the closest vertex and the second-closest vertex to an input point efficiently.

Chapter 4 Ch. 4 presents the details of *surface-reconstructing growing neural gas* (SGNG). At first, SGNG's immediate predecessor, the *growing self-reconstruction map* [41], is analyzed with special regard to its limitations that are caused by *competitive Hebbian learning* [93]. From this analysis, the requirements for SGNG are derived. Afterwards, the SGNG algorithm is described in detail. Pseudocode is presented for each part of the algorithm, as well as implementation details making SGNG an efficient reconstruction algorithm. Suitable learning parameters for SGNG are determined before testing it with a publicly available benchmark [14] and before comparing it to *screened Poisson surface reconstruction* by Kazhdan and Hoppe [78]. Further improvements included in SGNG are evaluated in separate experiments. Finally, SGNG is tested with real-world data ranging from original range data from the Stanford Scanning Repository [114] to point clouds extracted from images taken in Paris, France. Results indicate that SGNG improves upon its predecessors and achieves similar or even better performance than existing state-of-the-art reconstruction algorithms.

Chapter 5

Ch. 5 presents a texturing extension for SGNG. If SGNG is used to reconstruct objects from point clouds that have been extracted from images, then the images can be assigned as textures to the constructed triangles. That way, high-resolution visual details are added already to the initially coarse approximation of an object's shape that SGNG provides. At first, a straightforward way to learn surface color is presented. Afterwards, a more widely applicable algorithm is presented that selects suitable textures from the original images while reducing the number of occlusion artifacts to a minimum. Existing techniques derive visibility from the reconstructed geometry, but it is hard to guarantee in real-world applications that occluders are represented accurately in the input points and in the reconstruction. To overcome this, the presented extension learns the visibility information that is implicitly encoded in the input data. A visual evaluation using the data sets from Paris, France, reveals that SGNG with automatic texture assignment provides good visual quality and successfully reduces the number of occlusion artifacts as desired.

Chapter 6

Ch. 6 concludes this dissertation and relates the results and findings from the previous chapters to each other. Finally, further research directions are presented that have been determined during the work for this dissertation but that lie beyond the scope of this work.

Suggested reading paths

If this dissertation is read from Ch. 1 to 6 then it provides the reader with the motivation and the theoretical foundations for using *surface-reconstructing growing neural gas* to reconstruct textured 3D objects from unorganized point clouds in an iterative pipeline. However, the individual chapters are intended to be mostly self-contained so that a reader with a strong background in artificial neural networks based on Kohonen's self-organizing map may treat Ch. 2 merely as a reference while reading Ch. 4 and 5. In order to apply the texturing extension to other surface reconstruction algorithms, Ch. 5 provides a good starting point, given that the reader is familiar with the basic algorithm of SGNG. At present, the algorithm that creates a progressive mesh while learning, presented in Ch. 3, is restricted to be used in growing cell structures. Integrating it into SGNG has been beyond the scope of this dissertation. That chapter may therefore also serve as a starting point for future work.

1.3 RELATED WORK

Online surface reconstruction from unorganized point clouds using an artificial neural network with integrated texture mapping is the main focus of this dissertation. Therefore, the topics discussed in the subsequent chapters have a majority of the related work in common. It is presented here in order to provide a common context for the techniques examined in the remainder of this dissertation. Additional related work for specific subjects is presented when needed.

Since a large body of work has already been done, and since surface reconstruction is still an active field of research, only a small fraction of it can be covered in this dissertation. Nevertheless, several important techniques are presented in this section to give a representative overview of the field. The reader is referred to several available surveys and state of the art reports to find more information.

Surveys

A good starting point for getting a detailed overview of the research field of surface reconstruction is probably the recent state of the art report by Berger et al. [15]. It focuses on algorithms that approximate a surface from point clouds representing static objects. The report categorizes the discussed algorithms with respect to priors and the type of data imperfections that are addressed, as well as with respect to the format that the reconstructed surface is represented in.

The survey by Musialski et al. [99] focuses on techniques that are required for urban reconstruction. The survey categorizes the covered techniques with respect to their output type. It distinguishes algorithms that reconstruct individual buildings and, if applicable, corresponding semantics from algorithms for large-scale urban reconstruction up to massive city reconstruction. Besides presenting algorithms from these two categories, the survey reviews techniques for façade reconstruction. Finally, it provides a detailed explanation about stereo vision, structure from motion, multi-view stereo, and point cloud generation based on stereo vision. Therefore, that survey does not only provide related work but also the theoretical background for generating the data that is used as an input to the algorithms presented this dissertation.

The state of the art report by Attene et al. [11] reviews techniques for polygon mesh repairing. The individual techniques are grouped by defects that they resolve, like noise, holes, intersections, and degeneracies. Furthermore, the survey characterizes applications that generate meshes by the defects that they commonly produce, and it characterizes applications that use meshes, e.g., for visualization, by the defects that need to be avoided. It thus helps choosing appropriate mesh repairing algorithms when linking applications to form a processing pipeline. Especially the presented techniques for removing topological noise and for repairing non-manifold connectivity may serve as possible extensions to SGNG in order to improve the quality of early approximations from sparsely sampled input data. However, integrating them is left for future work.

The survey by Cazals and Giesen [25] covers techniques for surface reconstruction that are based on the Delaunay triangulation of the input point samples. The presented algorithms construct a surface that interpolates the input data. They pose strong requirements on the input data, but they may provide provable guarantees for the geometric and topological quality that the reconstruction of the surface can achieve. The survey categorizes the presented algorithms by the main approach they take, for instance, labeling the tetrahedra in the Delaunay triangulation of an input point set into tetrahedra that lie inside the volume or outside the volume that is enclosed by the surface to be reconstructed. The monograph by Dey [35] provides another overview on reconstruction techniques from the same field of research.

Implicit surfaces

In a very popular reconstruction approach, the surface is represented implicitly by a function that is fitted to the input data. Distance functions and indicator functions are most commonly used. The former assigns a value to each point that represents its distance to the original surface. The latter assigns a constant value to points inside an object and a different constant value to points outside an object. Afterwards the isosurface of a level set—often the zero-level set for distance functions—of this function is generated and triangulated, given that a triangle mesh is the intended output format. This output mesh is approximating the input data, and it serves as the reconstruction of the original object's surface. Variations of *marching cubes* by Lorensen and Cline [88] are commonly used for this purpose. However, it is known to produce many triangles that are irregular or close to degenerate.

Tangent planes proposed by Hoppe et al. [64] is a straightforward approach to estimate a distance function from a set of unorganized 3D points. For each input point **p** a tangent plane is estimated by applying principal component analysis to the group of *k* nearest neighbor points of **p**. Afterwards, the tangent planes are oriented consistently to create a piecewise linear approximation of the zero-level set of the desired distance function. Finally, the function is evaluated at the vertices of a prespecified cubical lattice for isosurface extraction.

Volumetric range image processing (VRIP) proposed by Curless and Levoy [30] uses range images as an input. From these range images a cumulative signed distance function and a cumulative weight function are computed. The functions are then evaluated on a discrete voxel grid, and an isosurface is extracted that represents the original object's surface. Finally, VRIP employs hole filling in order to create a watertight reconstruction.

Kazhdan [75] proposes to compute an indicator function in the frequency domain. Later, *Poisson surface reconstruction* (PSR) proposed by Kazhdan et al. [79] poses surface reconstruction as a spatial Poisson problem. For this purpose, the input points need to store at least an approximation of the oriented surface normals. A vector field is computed from the input points and the corresponding normals. This vector field is equal to the gradient of a smoothed indicator function that assigns a value of one to points inside the original object and a value of zero to points outside the object. An isosurface is extracted from the vector field by a variant of marching cubes that was adapted to an octree representation, leading to a watertight triangle mesh. An out-of-core implementation of PSR was proposed by Bolitho et al. [20] that allows for handling huge data sets.

Recently, PSR was refined to *screened Poisson surface reconstruction* (SPSR) by Kazhdan and Hoppe [78]. Additional positional constraints are included so that deviations from the input data are penalized by an additional energy term. That way, a mesh generated by SPSR represents the input data better than a mesh generated by PSR. Furthermore, SPSR improves the algorithmic complexity using hierarchical clustering and a conforming octree structure.

While VRIP, PSR, and SPSR use signed distance functions directly, the approach by Mullen et al. [97] creates an unsigned distance function as a starting point that is more robust to outliers and noise. The function is then discretized hierarchically in order to provide more detail close to the surface. A sign is estimated in a similar hierarchical way. At first, a coarse estimation is determined by ray shooting that is refined in regions close to the surface. Finally, the surface is extracted using *Delaunay refinement* by Boissonnat and Oudot [19].

The approach of Hornung and Kobbelt [67] also uses unsigned distances as a starting point. A confidence map is computed from the input points that, if evaluated on a voxel grid, yields the probability that the original surface passes through a given voxel. The surface is then extracted via a graph-cut proposed by the same authors [68].

Computational geometry

Another group of algorithms originated from computational geometry. These algorithms use the input data to partition the space that an original object is embedded in. A subset of this partition is then used to create a reconstruction of the original object.

In a very early paper two approaches are proposed by Boissonnat [18]. The first approach uses the fact that the surface of a three-dimensional object is two-dimensional. The algorithm creates an initial edge between a point and its closest neighbor. Afterwards, triangles are created iteratively by picking suitable points from the input data that have not been picked yet, and by connecting them to the existing contour edges. The second approach uses the Delaunay triangulation of the input points to start from. Afterwards, tetrahedra of the Delaunay triangulation are eliminated until all input points are located on the boundary of the resulting polyhedral shape.

Three-dimensional α-shapes proposed by Edelsbrunner and Mücke [43] also starts from the Delaunay triangulation of the set of input points. This algorithm generalizes the notion of planar *α-shapes* introduced earlier by Edelsbrunner et al. [44]. In the 3D approach, intuitively, the edges and faces of the tetrahedra are carved away by a ball of radius α. The ball reaches all positions for which no input point is located inside the ball. The remaining edges and faces represent the object to be reconstructed for a given value of α. For an actual implementation the valid α-intervals are precomputed for every edge and face of the Delaunay triangulation.

The *ball pivoting algorithm* proposed by Bernardini et al. [16] is closely related to *α-shapes* as it constructs a subset of an *α-shape*. At first, three input points are selected in such a way that a ball with a given radius that is touching the three points does not contain any other point. These input points are then connected to create a first seed triangle. Afterwards, the ball is pivoted around each contour edge. When it touches a new input point it is connected to the respective edge in order to create a new triangle. A new seed triangle is created once the ball has been pivoted around all edges. The algorithm terminates if no such triangle can be created.

The *crust* algorithm proposed by Amenta et al. [5] uses a Voronoi-filtered Delaunay triangulation in order to select the triangles that represent the original surface. At first, the Voronoi cells of the input points are computed. Afterwards, one (if the input point lies on the convex hull of the input data) or two poles are determined for each input point. These poles are represented by the vertices of a point's Voronoi cell that are farthest away from the point; one on the inside of the original surface, one on the outside. Finally, the Delaunay triangulation of the union of the set of input points and the set of poles is computed. The surface of the original object is then represented by the set of triangles of the Delaunay triangulation for which all three vertices are input points.

Power crust proposed by Amenta et al. [7] modifies this approach. At first, the poles are determined. Afterwards, the power diagram [12, 42] of the poles is constructed, which constitutes a Voronoi diagram that is weighted by the Euclidean distance of a pole to the corresponding input point. Finally, the surface of the original object is represented by the boundary separating the power diagram cells of the inner poles from the power diagram cells of the outer poles.

Both algorithms, *crust* and *power crust*, come alongside very elaborate theoretical guarantees and have been extensively refined. To mention only a few: *Cocone* by Amenta et al. [6] simplifies *crust* by eliminating some computation steps. *Tight Cocone* by Dey and Goswami [36] creates a watertight surface from a preliminary *cocone* reconstruction. Dey and Goswami [37] provide theoretical guarantees if the input data contains noise. *Eigencrust* by Kolluri et al. [84] improves the resistance to noise by using a spectral partitioning scheme.

Iterative pipeline All of the algorithms presented above, the ones using implicit functions and the ones from computational geometry, require the complete input data set to be available when they start. They therefore cannot provide any preview during scanning that can be used to direct further data acquisition. To give a first impression of the surface while scanning an object, points are extracted from matched depth images in a model-acquisition pipeline proposed by Rusinkiewicz et al. [109]. The points are then rendered using *QSplat rendering* proposed earlier by Rusinkiewicz and Levoy [108]. However, the final mesh is still constructed offline using vrip.

KinectFusion proposed by Newcombe et al. [100] basically provides a gpu implementation of the above approach: Arbitrary, complex indoor scenes are reconstructed in real-time using only a single Microsoft™ Kinect®. The acquired depth images are fused to create a single model. Unlike the above approach using frame-to-frame tracking, *KinectFusion* tracks against the full surface model acquired so far. While this approach provides real-time preview that is well suited to direct further data acquisition, it is restricted to depth images, whereas unorganized point clouds are a way more general type of input.

Learning

A third group of algorithms that are suitable for surface reconstruction are based on machine learning techniques. The presentation focuses on techniques that use a neural network, since *surface-reconstructing growing neural gas* that is presented in this dissertation is based upon those. However, a recent approach using a different learning scheme has been included for completeness.

Xiong et al. [133] recently proposed a surface reconstruction approach based on dictionary learning. A subset of input points that is selected by *Poisson disk sampling* [29] serves as an initial set of vertices for the mesh to be constructed. The connectivity and vertex positions are then learned iteratively, switching between connectivity learning, i.e., keeping the vertex positions fixed, and position learning, i.e., keeping the connectivity fixed. Optimization is performed by minimizing a sophisticated energy function including a point-to-mesh metric and a regularization term, and by enforcing a manifold constraint. The number of vertices remains fixed during learning. Thus, the approach cannot adapt to any modifications of the input data. Other dictionary-based techniques have successfully been applied to related tasks before. For instance, Gal et al. [55] use a dictionary of shape priors for surface reconstruction, and Wang et al. [124] create a dictionary representing sharp features for denoising.

Surface-reconstructing growing neural gas (sgng) is based on a family of closely related neural networks that were not designed for surface reconstruction in the first place, but that have been later applied to this task. These algorithms are very well suited for an iterative pipeline. They start learning as soon as the first input points are provided, and they refine their results while more input points become available.

The *self-organizing map* (som) proposed by Kohonen [81, 82, 83] is the most fundamental neural network that learns the positions of a prespecified number of neurons with a predefined connectivity from the input data. *Neural gas* (NG) [91, 92, 94] and the *topology representing network* (TRN) [93] proposed by Martinetz et al. added topology learning capabilities. *Growing cell structures* (GCS) proposed by Fritzke [49, 51, 53] added the capability to automatically learn the density of the neural network, i.e., the required number of neurons. Finally, *growing neural gas* (GNG) proposed by Fritzke [52, 53] combines both, learning the density and the topology. A detailed description of these fundamental neural networks including a taxonomy characterizing their capabilities is presented in Ch. 2.

Yu [134] was one of the first who applied Kohonen's SOM to surface reconstruction. The neurons of the network represent the vertices of the triangle mesh to be constructed. The edges that are connecting the neurons of the network represent the edges that are connecting the vertices of the triangle mesh. *Edge swap* operations [65] are added in order to improve the quality of the reconstructed mesh. Furthermore, a multiresolution-learning scheme is used. An initial approximation of the overall shape is determined with a low number of vertices. Once the desired accuracy is reached, each face is split into four smaller faces. Then, learning continues at a higher resolution, adding finer detail. That way, the number of iterations performed with detailed meshes, and thus with high computational cost, is reduced.

Várady et al. [119] applied Fritzke's GCS to free-form surface reconstruction and improved upon an earlier approach by Hoffmann and Varady [60] that used Kohonen's SOM. Instead of a triangle mesh, the neural network represents the control mesh of a NURBS or a Bézier surface. Thus, a new row or a new column is added to the control mesh in order to refine the network.

Neural meshes have been introduced by Ivrissimtzis et al. for surface reconstruction. Their first approach [70] modifies Fritzke's GCS by replacing the original neighbor learning scheme with tangential smoothing [117]. Furthermore, the *edge split* that is used to add vertices in active regions in GCS is replaced with a *vertex split* in order to balance the valences of the vertices. Finally, removal of inactive vertices is included. A later approach [71, 72, 73] introduces topology modifications by removing triangles that have become too large and by merging boundaries that are located close to each other. In addition to the above improvements, *neural mesh ensembles* [74] have been proposed in order to improve the reconstruction quality and the performance by employing ensemble learning. Based on the above *neural meshes*, Saleem [110] suggests a more efficient algorithm to locate active vertices for mesh refinement. Instead of using an activity counter, the vertices are ordered in a list, and active vertices are moved closer to the front of the list.

Smart growing cells (SGC) proposed by Annuth and Bohn [8] extend *neural meshes*. They introduce *aggressive cut out* for removing triangles in what they call degenerated regions of the mesh. SGC labels regions as degenerated in which the valence of a vertex is too high or that contain too many acute-angled triangles. After removing the triangles, the resulting boundary is repaired. Furthermore, SGC improves the boundary merging process of *neural meshes*. Finally, SGC adapts the density of the constructed mesh to the curvature of the original surface and improves the reconstruction quality of sharp features. A *tumble tree* proposed by Annuth and Bohn [9] is used to locate active vertices for mesh refinement efficiently. The same authors compare the performance of their SGC to an implementation based on Fritzke's GNG [10], and evaluate the parallelization potentials of the algorithms.

Melato et al. [95] evaluate the applicability of TRN by Martinetz and Schulten and Fritzke's GNG to surface reconstruction. In their implementation the vertex positions are optimized and the interconnecting edges are created during learning. The triangle faces are constructed in a post-processing step after learning has been finished.

Meshing growing neural gas proposed by Holdstein and Fischer [61] employs a very similar approach. At first, noisy input data is filtered by curvature analysis. Then, Fritzke's GNG is used to learn vertex positions and interconnecting edges. Afterwards, the required triangles are constructed during a post-processing step. Holdstein and Fischer furthermore propose to adaptively refine the constructed triangle mesh in user-selected regions.

The approach proposed by Fišer et al. [47] improves the nearest-neighbor search that is required in Fritzke's GNG. A growing uniform grid is used to reduce the computational complexity of the algorithm. Furthermore, handling of a vertex' activity is improved in order to reduce reconstruction times.

The *growing self-reconstruction map* (GSRM) proposed by do Rêgo et al. [39, 40, 41] is the immediate predecessor of SGNG that is presented in this dissertation. GSRM improves upon Fritzke's GNG by creating some triangles during learning. GSRM aims at avoiding self-intersections and non-manifold edges with more than two adjacent triangles. Furthermore, obtuse triangles are removed once they are detected in order to improve the quality of the reconstructed mesh. However, GSRM still requires a post-processing step to complete learning the topology and to triangulate remaining holes. GSRM is covered in detail in Sec. 2.8. The *growing self-organizing surface map* proposed by DalleMole et al. [31, 32, 33] is closely related to GSRM. However, the former implements a different connection-learning rule.

Recently, Orts-Escolano et al. [101, 102] proposed an algorithm that extends GSRM. In addition to creating only single triangles adjacent to an edge, the algorithm triangulates quadrangles and pentagons once they are detected during learning. While it leaves fewer holes untriangulated, their approach still requires additional processing steps to finalize the topology once learning has been finished. However, some higher-order polygons cannot be closed at all. Nevertheless, Orts-Escolano et al. integrate a straightforward method to approximate color and normal information.

Learning algorithms that rely on post-processing to complete the triangle mesh are not very well suited for an iterative pipeline where data acquisition, reconstruction, and visualization are executed in parallel: Learning has to be interrupted repeatedly for post-processing while acquiring data. Thus, reconstruction gets costly. Iterative algorithms that keep the number of vertices fixed can provide visual feedback while reconstructing. However, reconstruction has to be restarted from scratch whenever the input data was modified. SGNG that is presented in this dissertation overcomes these shortcomings in order to provide visual feedback while acquiring further data.

Texture assignment

None of the existing reconstruction techniques support texture assignment directly, although point clouds with registered images are available by now: Points can be extracted from images, for instance, using multi-view stereo [54] or *structure from motion* (SFM) [130]. Furthermore, modern 3D scanners provide RGB images that are registered to the points. SGNG automatically assigns the image as a texture to a triangle that provides the most perpendicular view of the triangle and that most likely shows the unoccluded triangle. Therefore, SGNG even allows for incremental textured preview while acquiring data, reducing the number of occlusion artifacts to a minimum. Texture assignment is integrated into SGNG learning. Other related techniques exist, but would have to be executed after reconstruction has finished.

Lempitsky and Ivanov [85] propose an approach for assigning images as textures to the triangles of the mesh using mesh-based *Markov random field* (MRF) energy optimization. Suitable texture coordinates are obtained by projecting the triangles into the images. An energy function is used to select the image as a texture that provides the most perpendicular view of a triangle and that reduces the noticeability of seams between different textures on adjacent triangles. Minimization is done via α-expansion graph cuts [21]. The proposed technique reduces the overall length of texture seams, placing the seams into regions where only little texture misalignment occurs.

Sinha et al. [113] use a very similar approach, but define the MRF on a texel grid. A more sophisticated energy function is used in their approach that also considers the depth values encoded in the point cloud in order to reduce the number of artifacts due to occluders that are not reconstructed accurately. Furthermore, a user may label image regions as suitable or unsuitable for texturing by brushing.

Abdelhafiz [1] proposes an approach that selects the image as a texture that provides the most perpendicular view of a triangle and in which the projected area of the triangle is largest. Two steps of occlusion handling are applied. The first uses the depth information acquired from the reconstructed object in order to determine which triangles are visible in an image. The second relies on color similarities.

The approach by Gal et al. [56] improves upon the approach by Lempitsky and Ivanov [85]. The former adds a set of local image transformations to the search space for optimization. That way, the number of noticeable artifacts due to misalignments and inaccurate reconstruction is reduced to a minimum. However, the approach does not address any artifacts that are caused by occlusion.

Musialski et al. [98] propose a system for generating façade ortho-textures from perspective images taken with hand-held cameras. The approach automatically constructs façade planes and aligns the images by SFM. User input is required to define the extent of each façade. Some occlusion artifacts are avoided automatically by using the depth information obtained from SFM. Similar to the approach by Sinha et al. [113], a user can interactively remove image regions that are unsuitable for texturing.

Dellepiane et al. [34] warp the images locally to compensate for small inaccuracies in reconstruction and texture alignment. Their approach computes the optical flow between overlapping images. Since the camera positions vary too much for direct computation, the algorithm performs pair-wise computations, projecting one image onto the reconstructed object and then into the image space of the second image. Afterwards, a displacement for the pixels in the first image is computed in the image space of the second image. The displacement is then projected via the reconstructed object into the image space of the first image. However, occlusions are not handled.

Birsak et al. [17] propose a pipeline for assigning photos as textures to a triangle mesh. Their approach improves upon the algorithm of Lempitsky and Ivanov [85] including the masks for the images that have been proposed by Callieri et al. [23]. These masks allow for selecting images with better quality by considering, for instance, a pixel's distance to the image borders or a pixel's focusing. Furthermore, a user can provide a stencil mask, excluding parts of the photos. In addition to that, the approach handles occlusion by first rendering a depth map using the reconstructed object. Afterwards, an image is considered as a texture for a triangle only if the triangle is visible in the image according to the depth map. This approach is similar to the one proposed by Chen et al. [26] and requires that all occluders are reconstructed accurately.

In addition to the texture assignment and occlusion handling presented above, each of the approaches employs techniques that reduce the noticeability of texture seams by blending and leveling. Integrating them into SGNG is beyond the scope of this dissertation and has therefore been left for future work.

1.4 SCIENTIFIC PUBLICATIONS

The contributions presented in this dissertation are based on the following publications:

A first sketch of the intended processing pipeline in which SGNG will be used and a virtual reality-based simulation framework for the pipeline have been presented at the joint virtual reality conference of EGVE - ICAT - EuroVR (JVRC'10) [115]. Prototypic results of iterative surface reconstruction from unorganized point clouds using a *self-organizing map* in the intended pipeline have been presented at the international symposium on virtual reality innovations (ISVRI'11) [116].

The evaluation and the relationship of *growing cell structures* (GCS) and a *progressive mesh* (PM) as well as the intuitive approach for letting GCS learn a PM that is described in Ch. 3 have been presented at the 33rd annual conference of the European association for computer graphics (EG'12) [121].

A first version of SGNG, the reconstruction algorithm presented in Ch. 4, has been published as a technical report by the department of computer science of the University of Muenster [122]. A prototypic sketch of the texturing extension presented in Ch. 5 has been presented at the first Eurographics workshop on urban data modelling and visualisation (UDMV'13) [123]. The complete version of SGNG as well as the improved texturing extension will be presented at the shape modeling international conference (SMI'15). The corresponding article is published in Computers & Graphics [120].

2 FUNDAMENTAL ARTIFICIAL NEURAL NETWORKS

Surface-reconstructing growing neural gas (SGNG), the reconstruction algorithm that is proposed in this dissertation, is intended to fit seamlessly into an iterative processing pipeline where scanning, reconstruction, and visualization are executed in parallel. SGNG is based on an artificial neural network that allows for continuous online learning.

In order to make this dissertation self-contained, the learning algorithms and the capabilities and characteristics of a family of related neural networks are presented in this chapter: The fundamental algorithm is described on the basis of a very early approach. Each network discussed afterwards adds a certain feature to the learning algorithm up to the immediate predecessor of SGNG. This chapter summarizes the information from the original publications in a consistent way in order to provide the theoretical background for the subsequent chapters.

2.1 HISTORY AND TAXONOMY

1981: SOM

Kohonen's *self-organizing map* (SOM) [81, 82, 83] proposed in 1981 is the most fundamental type of the neural networks that are used for learning-based surface reconstruction, and that are summarized in this chapter. SOM iteratively learns the positions of a fixed number of vertices that are connected in a mesh with a prespecified topology. Yu [134] was the first to apply SOM for surface reconstruction. He proposes edge swap and multiresolution learning to make the algorithm more effective and more efficient.

1991: NG

In order to also learn the topology of the mesh Martinetz et al. [91, 92, 94] proposed *neural gas* (NG) in 1991. The vertex positions are learned in a similar way as in SOM. However, after the position learning phase has been finished, edges are created in a post-processing step according to a *competitive Hebbian learning* (CHL) rule.

1992: GCS

Shortly after NG was developed, Fritzke [49, 51, 53] proposed *growing cell structures* (GCS) in 1992. GCS constructs a mesh from an initial,

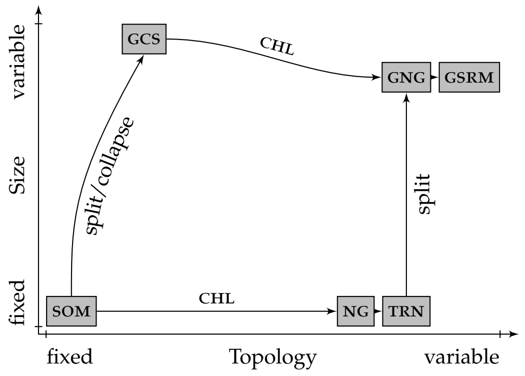


Figure 2.1: The relation of the fundamental neural networks and their capabilities to learn the topology and the network size.

k-dimensional simplex, i.e., a line ($k = 1$), a triangle ($k = 2$), or a tetrahedron ($k = 3$), depending on the surface to be reconstructed. The homeomorphic type of the constructed mesh is therefore prespecified, and as in SOM it will not be changed during learning. In contrast to SOM, GCS is able to adapt the number of vertices to the distribution of the input data: Vertices are added by an *edge split* or removed by an *edge collapse*. Such a growing scheme allows for simplified learning rules for the vertex positions. GCS was used later for surface reconstruction among others by Várady et al. [119], by Ivrissimtzis et al. [70], and by Annuth and Bohn [8].

1994: TRN In 1994 Martinetz and Schulten [93] refined their NG to the *topology representing network* (TRN) by combining both position and topology learning into the iterative phase. That way the entire set of edges is created during learning without relying on post-processing steps.

1995: GNG After having developed the growing scheme for GCS, Fritzke [52, 53] combined it with the capabilities of the recently developed TRN into his *growing neural gas* (GNG) in 1995. He proposed a network that is able to construct a mesh with both the topology and the number of vertices learned from the input data. GNG was later used for surface reconstruction among others by Holdstein and Fischer [61], and by do Rêgo et al. [39, 40, 41]. The former create the desired triangles during a post-processing step after GNG learning has finished, similar to the post-processing step used in NG.

2010: GSRM In 2010 the *growing self-reconstruction map* (GSRM) was proposed by do Rêgo et al. [39, 40, 41]. GSRM is used as a basis for the development of SGNG. The former creates some of the desired triangles during the learning process. However, GSRM cannot create all triangles during learning as the underlying NG, TRN, and GNG, and thus still requires a post-processing step to complete the triangle mesh. Therefore GSRM is not particularly well suited for continuous online surface reconstruction.

Fig. 2.1 gives an overview of the relationships between the above artificial neural networks and their capabilities to learn the topology and the network size, i.e., the number of vertices. Since GCS modifies vertex connectivity while learning, it is considered to be slightly more versatile in terms of the constructed topology than SOM. Since TRN creates edges and GSRM creates triangles during the main learning loop, they are considered to be slightly more versatile in terms of the constructed topology than NG or GNG, respectively. GNG and GSRM are considered to be slightly less versatile in terms of the network size than GCS, since both rely on the topology learning steps to remove superfluous vertices instead of addressing them explicitly.

2.2 THE BASIC RECONSTRUCTION ALGORITHM

The artificial neural networks that are outlined in the subsequent sections learn the positions of a set $\mathcal{V}$ of neural units or vertices from a set $\mathcal{P}$ of unorganized input points. The vertices are connected by a set $\mathcal{E}$ of edges according to a prespecified or learned topology to create a mesh. Some of the networks even create a set $\mathcal{F}$ of triangles.

All the networks share the same basic reconstruction algorithm: In the beginning the network is initialized, i.e., the initial sets of vertices, edges and, if applicable, faces are created. Afterwards, learning is iterated in a loop for a prespecified number $\mathrm{t}_{\mathrm{max}}$ of iterations or until a predefined convergence criterion is met. In each iteration t an input point $\mathbf{p}_{\xi_t}$ is selected randomly from the set $\mathcal{P}$ of input points. Then, the neural network adapts the vertex positions and, if the network allows for it, the topology to the selected input point according to specific learning rules. If the neural network relies on a post-processing step, it is applied after the learning loop has finished.

```
$\langle$Reconstruction$\rangle \equiv$
  $\langle$Initialization$\rangle$
  for each $t \in \{1, 2, \ldots, \mathrm{t}_{\mathrm{max}}\}$ do
      randomly select $\mathbf{p}_{\xi_t} \in \mathcal{P}$
      $\langle$Updates according to $\mathbf{p}_{\xi_t}\rangle$
  end for each
  $\langle$Post-processing step (if applicable)$\rangle$
```

The individual reconstruction algorithms of all networks are presented in detail in the subsequent subsections.

Vertex positions, edges, sets, etc. are initialized in the beginning and updated in each iteration of the learning algorithm. In order to keep the notation concise the assignment operator $:=$ is used for this purpose. The instruction

$$x := x + y$$

calculates $x + y$ and assigns the result to x afterwards, similar to `x=x+y` in a programming language like, e.g., C.

The artificial neural networks described in this chapter are generally able to operate in arbitrary dimensional spaces. However, this

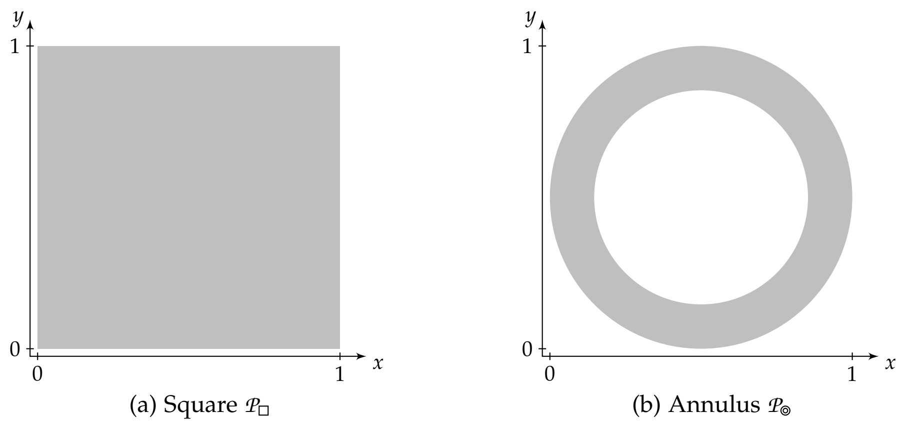

y
1
0
0
1
x
(a) Square $\mathcal{P}_{\square}$
y
1
0
0
1
x
(b) Annulus $\mathcal{P}_{\circledcirc}$

Figure 2.2: Input point sets used in the examples (gray area).

work focuses on reconstruction of 2D surfaces embedded in 3D space. The following descriptions therefore use 3D positions. Nevertheless, extensions to higher dimensional spaces are straightforward.

Examples The characteristics of each neural network is illustrated with two examples that are using input data sampled from surfaces with different genera. For the sake of clarity of the figures only input points of the form $\begin{bmatrix} x & y & 0 \end{bmatrix}^\top \in \mathbb{R}^3$ are used. In the first example the input points are sampled uniformly at random from a square (Fig. 2.2(a))

$$\mathcal{P}_{\square} = \left\{ \begin{bmatrix} x & y & 0 \end{bmatrix}^\top \mid x, y \in [0,1] \right\} \quad . \tag{2.1}$$

In the second example the input points are sampled uniformly—with respect to the surface area—at random from an annulus (Fig. 2.2(b))

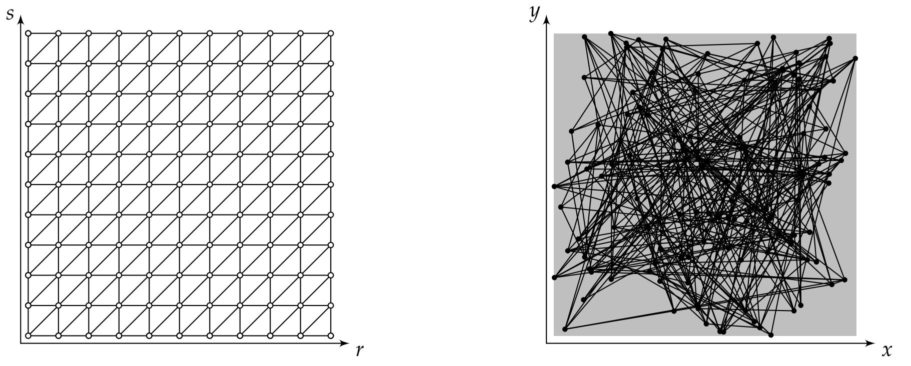
$$\mathcal{P}_{\circledcirc} = \left\{ \mathbf{p} \in \mathcal{P}_{\square} \mid \left\| \mathbf{p} - \begin{bmatrix} 0.5 & 0.5 & 0 \end{bmatrix}^\top \right\| \in [\sqrt{0.125}, 0.5] \right\} \quad . \tag{2.2}$$

2.3 SELF-ORGANIZING MAP

Kohonen's *self-organizing map* (SOM) [81, 82, 83] is the most fundamental artificial neural network that is related to the final reconstruction algorithm proposed in this dissertation. The reconstruction algorithm consists of two phases.

$\langle$*SOM reconstruction*$\rangle \equiv$
  $\langle$*SOM initialization* $\rightarrow$ p. 23$\rangle$
  $\langle$*SOM learning* $\rightarrow$ p. 24$\rangle$

SOM initialization During initialization a set $\mathcal{V}$ of neural units $v_{i,j} \in \mathcal{V}$ is created that are arranged in a rectangular grid of h rows and w columns (Fig. 2.3(a)). In order to create a mesh representing a 2D surface $h, w \geq 2$. To each unit $v_{i,j} \in \mathcal{V}$ its normalized 2D location $\mathbf{u}_{i,j} = \begin{bmatrix} r_{i,j} & s_{i,j} \end{bmatrix}^\top \in [0,1]^2$ in the neural network is assigned. These 2D locations will not be modified during learning. Since each unit of the neural network represents a



(a) Locations and connectivity of the units in the neural network.

(b) Initial positions and connectivity of the vertices in the mesh.

Figure 2.3: Initial configuration of SOM reconstructing a square (gray): The individual units (◦) of the neural network are arranged in a regular grid inside the network (a). The vertices (•) of the initial triangle mesh are placed at the positions of randomly selected input points (b).

vertex in the mesh to be constructed, a 3D position $\mathbf{v}_{i,j} \in \mathbb{R}^3$—in the space into which the original surface is embedded—is assigned to each unit. The vertices of the initial mesh share the connectivity of the neural units in the neural network: An edge that is connecting a pair of neural units is also connecting the pair of vertices that are represented by the units. The positions $\mathbf{v}_{i,j}$ of the vertices are initialized to the positions of randomly selected input points (Fig. 2.3(b)). The 3D positions $\mathbf{v}_{i,j}$ will be optimized during learning.

⟨*SOM initialization*⟩ ≡ ← p. 22

$$\mathcal{V} := \{v_{i,j} \mid i = 1, \dots, h \ , \ j = 1, \dots, w\} \ , \ i, j, h, w \in \mathbb{N}$$

$$\mathbf{u} : \mathcal{V} \mapsto \mathbb{R}^2 \quad \text{where} \quad \mathbf{u}(v_{i,j}) = \mathbf{u}_{i,j} := \begin{bmatrix} \frac{j-1}{w-1} & \frac{i-1}{h-1} \end{bmatrix}^\top$$

$$\mathbf{v} : \mathcal{V} \mapsto \mathbb{R}^3 \quad \text{where} \quad \mathbf{v}(v_{i,j}) = \mathbf{v}_{i,j} := \mathbf{p}_{\xi_{i,j}} \in \mathcal{P}$$

The shorthands $\mathbf{u}_{i,j}$ and $\mathbf{v}_{i,j}$ are used instead of $\mathbf{u}(v_{i,j})$ and $\mathbf{v}(v_{i,j})$, respectively, in the remainder of this work.

To define the topology of the neural network and thus the mesh to be constructed, a set $\mathcal{E}$ of horizontal, vertical, and diagonal edges is created in the neural network (Fig. 2.3(a)).

⟨*SOM initialization*⟩ +≡ ← p. 22

$$\begin{aligned} \mathcal{E} := \{(v_{k,l}\,, v_{k,l+1}) \mid v_{k,l}\,, v_{k,l+1} \in \mathcal{V}\} &\cup \\ \{(v_{k,l}\,, v_{k+1,l}) \mid v_{k,l}\,, v_{k+1,l} \in \mathcal{V}\} &\cup \\ \{(v_{k,l}\,, v_{k+1,l+1}) \mid v_{k,l}\,, v_{k+1,l+1} \in \mathcal{V}\}& \end{aligned}$$

Since the vertices in the constructed mesh share the connectivity of the neural units, the initial triangle mesh is randomly folded (Fig. 2.3(b)).

SOM learning

Online learning is iterated in a loop for a prespecified number $t_{\max}$ of iterations. Alternatively, a predefined convergence criterion can be

Table 2.1: Learning parameters used in the SOM examples.

Parameter		Value
Step size	$\epsilon_i \ldots \epsilon_f$	$0.5 \ldots 0.005$
Standard deviation of lateral influence	$\sigma_i \ldots \sigma_f$	$1.0 \ldots 0.01$
Maximum number of iterations	$\mathrm{t_{max}}$	30 000

used, for instance, a prespecified maximal mean distance between the input points and the vertex positions.

In each iteration t of SOM learning an input point $\mathbf{p}_{\xi_t}$ is selected randomly from the set $\mathcal{P} \subset \mathbb{R}^3$ of input points. Afterwards, the 3D positions assigned to the neural units are adapted to $\mathbf{p}_{\xi_t}$.

```
⟨SOM learning⟩ ≡                                              ← p. 22
    for each $t \in \{1, 2, \ldots, \mathrm{t_{max}}\}$ do
        randomly select $\mathbf{p}_{\xi_t} \in \mathcal{P}$
        ⟨SOM position updates → p. 24⟩
    end for each
```

SOM position updates For the selected input point $\mathbf{p}_{\xi_t}$ the best matching, i.e., closest, unit $v_\mathrm{b} \in \mathcal{V}$ with respect to the assigned 3D position $\mathbf{v}_\mathrm{b}$ and the ℓ^2-norm $\|\cdot\|$ is determined. Then, the 3D position $\mathbf{v}_i$ of each unit $v_i \in \mathcal{V}$ is adapted to $\mathbf{p}_{\xi_t}$.

```
⟨SOM position updates⟩ ≡                                      ← p. 24
    $v_\mathrm{b} = \arg\min_{v_i \in \mathcal{V}} \|\mathbf{v}_i - \mathbf{p}_{\xi_t}\|$
    $\mathbf{v}_i := \mathbf{v}_i - \epsilon(t) \cdot h_\sigma(t, v_i, v_\mathrm{b}) \cdot (\mathbf{v}_i - \mathbf{p}_{\xi_t}) \quad , \ \forall v_i \in \mathcal{V}$
```

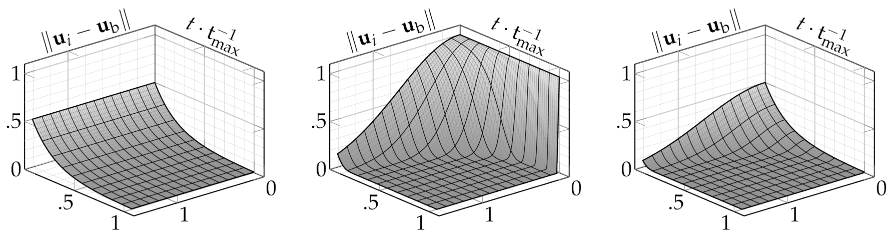
This update rule uses a time dependent step size specifying the overall adaptation rate. To be consistent with the description of NG by Martinetz et al. [92, 94] in Sec. 2.4 the following time dependence is used here:

$$\epsilon : \mathbb{N} \to [0,1] \quad , \quad \text{where} \quad \epsilon(t) = \epsilon_i \cdot \left(\frac{\epsilon_f}{\epsilon_i}\right)^{t \cdot \mathrm{t_{max}^{-1}}} \quad .$$

Initially, $\epsilon(t = 0) = \epsilon_i$. It decreases monotonically over time $t \in \mathbb{N}$ in such a way that $\epsilon(t = \mathrm{t_{max}}) = \epsilon_f$. Furthermore, the update rule of SOM uses a time dependent neighborhood function

$$h_\sigma : \mathbb{N} \times \mathcal{V} \times \mathcal{V} \to [0,1] \quad ,$$
$$\text{where} \quad h_\sigma(t, v_i, v_\mathrm{b}) = \exp\left(-\frac{\|\mathbf{u}_i - \mathbf{u}_\mathrm{b}\|^2}{2\sigma(t)^2}\right)$$

specifying a lateral influence among the neural units. The lateral influence is related to the distance of a unit v_i to v_b in the neural network. It is modeled by a Gaussian [83, 94] centered at the 2D



(a) $\epsilon(t)$ (b) $h_\sigma(t, v_i, v_b)$ (c) $\epsilon(t) \cdot h_\sigma(t, v_i, v_b)$

Figure 2.4: Step size (a), lateral influence (b), and combined learning rate (c) of SOM (Tab. 2.1), plotted against the distance of v_i to v_b and the normalized number of iterations.

location $\mathbf{u}_b$ of the best matching unit v_b in the network, with a time dependent standard deviation

$$\sigma : \mathbb{N} \to \mathbb{R} \quad , \quad \text{where} \quad \sigma(t) = \sigma_i \cdot \left(\frac{\sigma_f}{\sigma_i}\right)^{t \cdot t_{\max}^{-1}} \quad . \tag{2.3}$$

Thus, $h_\sigma(t, v_i, v_b) = 1 \iff v_i = v_b$, and $0 \leq h_\sigma(t, v_i, v_b) < 1$ otherwise. Initially, $\sigma(t = 0) = \sigma_i$. It decreases monotonically over time $t \in \mathbb{N}$ in such a way that $\sigma(t = t_{\max}) = \sigma_f$. Since the step size $\epsilon(t)$, the value of the lateral influence $h_\sigma(t, v_i, v_b)$, and the radius around the best matching unit in which h_σ has an effect are initially large, SOM can create an ordered map, i.e., unfold the mesh. Since all three monotonically decrease over time, SOM initially performs large updates that get smaller the longer the algorithm is running.

Example for SOM *learning*

Fig. 2.3 shows an example of an initial SOM configuration. The network contains 121 connected neural units that are arranged in a regular 11 by 11 grid. The learning parameters that are used in the reconstruction examples are set to the values that Martinetz and Schulten [93] used (Tab. 2.1), in order to be comparable with the next two neural networks. The initial standard deviation of the lateral influence was set to half the value that Martinetz and Schulten used in order to emphasize the characteristics of SOM in the examples.

Fig. 2.4 shows plots for the step size, the lateral influence, and the combined learning rate. In order to give an impression of the progression of the respective function in the neural network and over time, the value of the respective function is plotted on the z-axis against the distance of a unit v_i to the best matching unit v_b on the x-axis, and the number t of iterations normalized to the maximum number $t_{\max}$ of iterations on the y-axis. The distance of the units is computed from the 2D locations $\mathbf{u}_i$ and $\mathbf{u}_b$ of the units in the neural network. As required for the creation of an ordered map from the random initialization, all three functions decrease monotonically over time. The step size $\epsilon(t)$ is constant over 2D distance to v_b, but the lateral influence $h_\sigma(t, v_i, v_b)$

and thus the combined learning rate monotonically decrease over distance.

In the first example the input points $\mathcal{P} \subset \mathcal{P}_{\square}$ (Eq. 2.1) are drawn uniformly at random from a square. Fig. 2.5 shows the resulting mesh after different numbers of iterations. It can easily be seen that the constructed mesh shrinks but unfolds itself during the first iterations (Fig. 2.5(a)–(d)) due to a wide lateral influence. Afterwards, while the lateral influence is narrowing down to only the best matching unit, the vertices spread out (Fig. 2.5(e)–(g)) until they are fairly evenly distributed across the input square (Fig. 2.5(h)).

Relation to k-means clustering

If $\sigma(t) \to 0$ and thus $h_\sigma(t, v_i, v_b) \to \delta_{v_j, v_b}$ the Kronecker delta, then SOM learning will become equivalent to an online, stochastic variant of the k-means clustering algorithm [87, 90]. In that case, the learning rules of SOM minimize the overall position error of the vertices in the mesh by a stochastic gradient descent on an energy function

$$E = \sum_{v_i \in \mathcal{V}} \sum_{\mathbf{p}_k \in \mathcal{P}_{\mathbf{v}_i}} \|\mathbf{v}_i - \mathbf{p}_k\|^2 \quad . \tag{2.4}$$

Here $\mathcal{P}_{\mathbf{v}_i} \subseteq \mathcal{P}$ contains all input points that are located in the Voronoi cell of $\mathbf{v}_i$, i.e., all input points that are closer to the 3D position $\mathbf{v}_i$ assigned to vertex v_i than to the 3D position assigned to any other vertex with respect to the ℓ^2-norm. Unfortunately, the above energy function has several local minima leading to folded meshes.

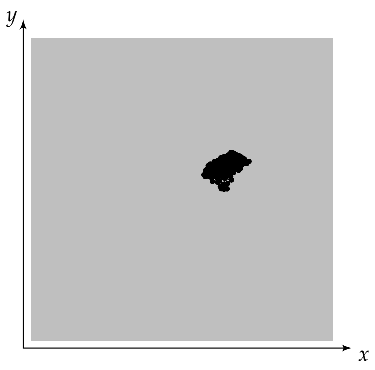
Lateral influence leads to ordered map

In order to create an ordered map, i.e., a mesh that is not folded, and thus to avoid the undesired local minima, the units in the neural network may not be affected independently of each other as Kohonen [83] noted. Therefore, SOM accounts for the topological relation among the neural units by using an initially large but monotonically decreasing lateral influence $h_\sigma(t, v_i, v_b)$. Due to this lateral influence, the learning rules cannot be described as a gradient descent on a single energy function anymore—neither one like (Eq. 2.4) nor any other one—as Erwin et al. [45] proved.

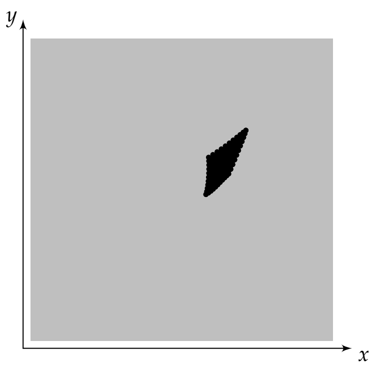
In fact, using a wide lateral influence with $\sigma(t) \neq 0$ in early iterations is crucial for SOM, since the 3D positions are initialized randomly. Otherwise, with $h_\sigma(t, v_i, v_b) \neq 0$ only for $v_i = v_b$ in all iterations, the mesh will not be unfolded at all (Fig. 2.6(a)). If a constant but narrow lateral influence is used in such a way that only the best matching unit and its directly connected neighbors are affected, the mesh will be unfolded only partially by SOM learning (Fig. 2.6(b)). Nevertheless, the latter concept is used later in algorithms that induce an initial ordering of the vertices by construction (Sec. 2.6–2.8).

Predefined topology

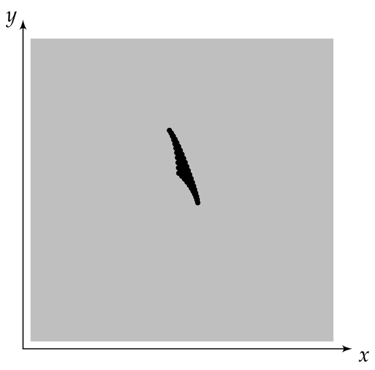
The topology of the surface that is reconstructed by SOM is predefined by the initial topology of the network. It is not modified during learning. For an illustration the input points $\mathcal{P} \subset \mathcal{P}_{\circledcirc}$ (Eq. 2.2) are drawn uniformly at random from an annulus in the second example. Fig. 2.7 shows the resulting mesh after different numbers of iterations.



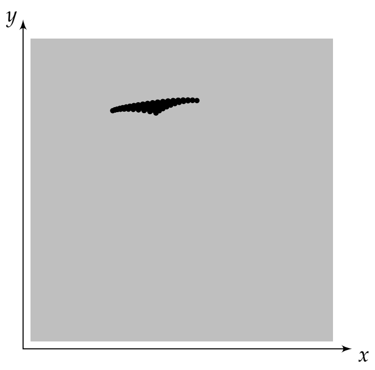
(a) $t = 10$ iterations



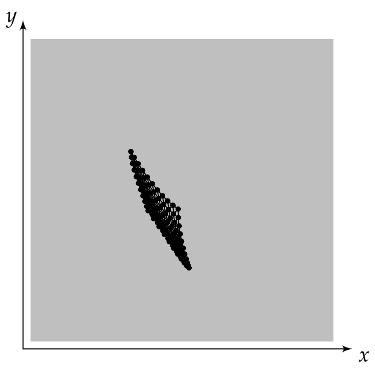
(b) $t = 30$ iterations



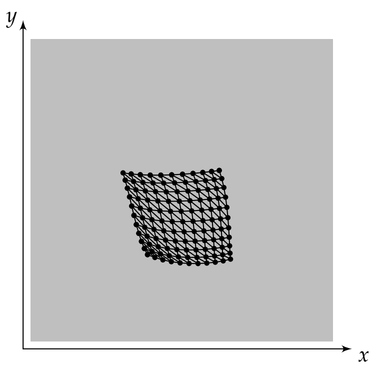
(c) $t = 100$ iterations



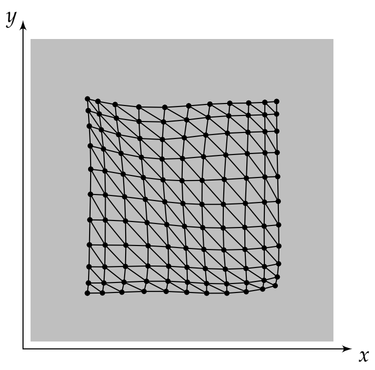
(d) $t = 300$ iterations



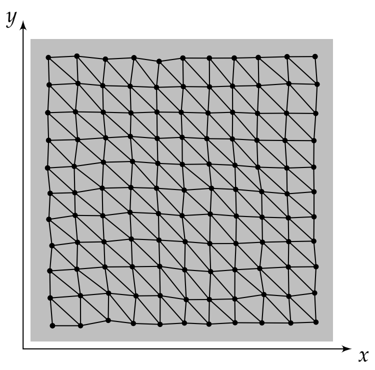
(e) $t = 1000$ iterations



(f) $t = 3000$ iterations



(g) $t = 10\,000$ iterations



(h) $t = 30\,000$ iterations

Figure 2.5: SOM reconstructing a square.

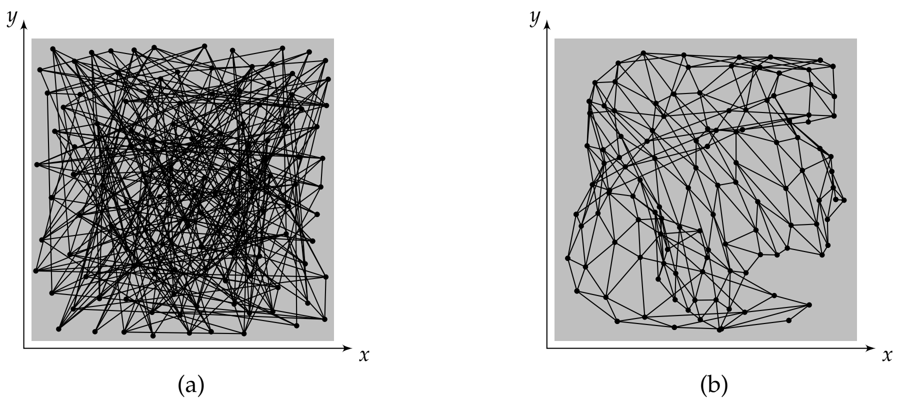


Figure 2.6: Meshes created by 30 000 iterations of SOM learning using different lateral influences: An input point is affecting (a) only the best matching unit, (b) only the best matching unit and its direct topological neighbors.

As in the previous example the mesh initially shrinks (Fig. 2.7(a)–(e)) and eventually unfolds itself (Fig. 2.7(f), (g)). Finally, nearly all vertices are located on or close to the input annulus (Fig. 2.7(h)). However, it can easily be seen that SOM is unable to reconstruct the topology of the input data: The learning algorithm yields a mesh that bridges the hole of the annulus.

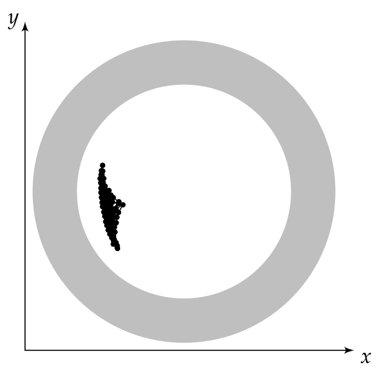
2.4 NEURAL GAS WITH SUBSEQUENT COMPETITIVE HEBBIAN LEARNING

The iterative updates of the vertex positions in SOM yield very good results if the predefined topology of the network matches the topology of the input data. To learn arbitrary topologies Martinetz et al. [91, 92, 94] proposed *neural gas* (NG) that creates all of the edges, i.e., the topology, in a separate post-processing step using *competitive Hebbian learning* (CHL). The basic reconstruction algorithm of NG is similar to that of Kohonen's SOM.

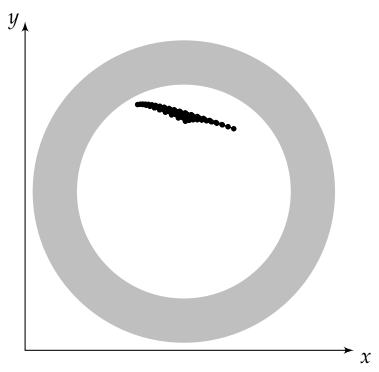
```
⟨NG reconstruction⟩ ≡
  ⟨NG initialization   → p. 30⟩
  ⟨NG position learning   → p. 30⟩
  ⟨NG topology learning   → p. 32⟩
```

In NG—and in all the following networks—the neural units and the vertices of the mesh to be constructed do not need to be treated separately anymore. Furthermore, the 2D location of a unit in the network is not required. That way, notation gets simplified in such a way that $\mathbf{v}_i$ simultaneously refers to the neural unit and the related vertex position. Thus, from now on $\mathcal{V} \subset \mathbb{R}^3$ for the set of vertices.

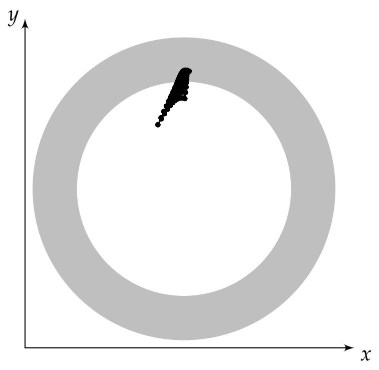
NG initialization During initialization a set $\mathcal{V}$ of n vertices $\mathbf{v}_i \in \mathcal{V} \subset \mathbb{R}^3$ is created. In order to create a mesh representing a 2D surface, $n \geq 3$. The positions



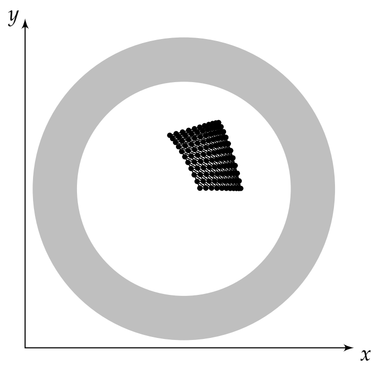
(a) $t = 10$ iterations



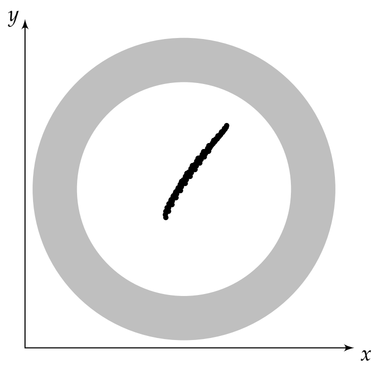
(b) $t = 30$ iterations



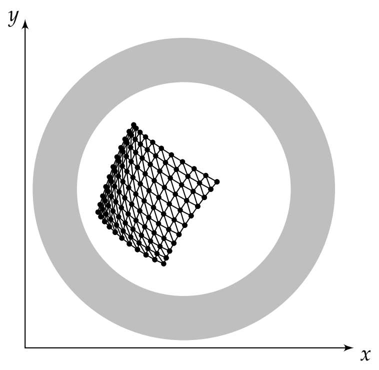
(c) $t = 100$ iterations



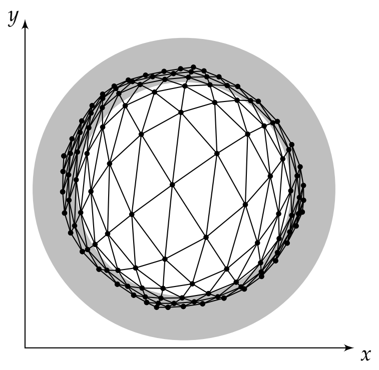
(d) $t = 300$ iterations



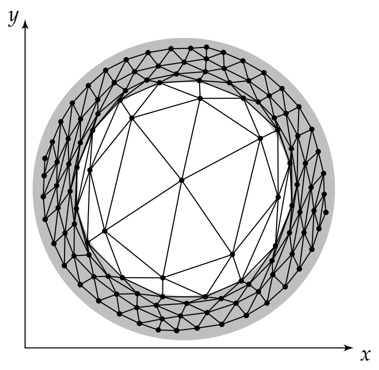
(e) $t = 1000$ iterations



(f) $t = 3000$ iterations



(g) $t = 10\,000$ iterations



(h) $t = 30\,000$ iterations

Figure 2.7: Som reconstructing an annulus.

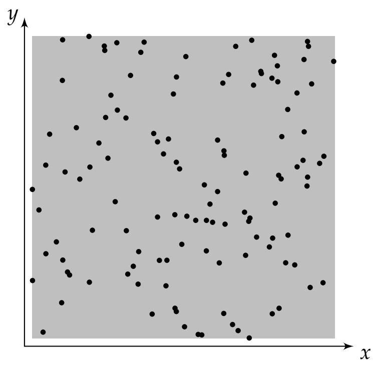


Figure 2.8: Initial configuration of NG reconstructing a square.

$\mathbf{v}_i$ of the vertices are initialized to the positions of n randomly selected input points (Fig. 2.8). The vertex positions will be optimized during learning, but edges will be created only in the post-processing step after position learning has finished.

```
⟨NG initialization⟩ ≡                                        ← pp. 28, 35
  $\mathcal{V} := \{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$ ,  $\mathbf{v}_i := \mathbf{p}_{\xi_i} \in \mathcal{P}$ ,  $i, n \in \mathbb{N}$ ,  $n \geq 3$
  $\mathcal{E} := \emptyset$
```

NG position learning As for SOM, online learning is iterated in a loop for a prespecified number $t_{\max}$ of iterations. Alternatively, a predefined convergence criterion can be used, for instance, a prespecified maximal mean distance between the input points and the vertex positions. However, the update rules for vertex positions of NG differ from that of SOM.

In each iteration t of NG learning an input point $\mathbf{p}_{\xi_t}$ is selected randomly from the set $\mathcal{P} \subset \mathbb{R}^3$ of input points. Afterwards, the vertex positions are adapted to the selected input point.

```
⟨NG position learning⟩ ≡                                     ← p. 28
  for each $t \in \{1, 2, \ldots, t_{\max}\}$ do
      randomly select $\mathbf{p}_{\xi_t} \in \mathcal{P}$

      ⟨NG position updates  → p. 31⟩
  end for each
```

NG position updates For the selected input point $\mathbf{p}_{\xi_t}$ a neighborhood ranking $(\mathcal{V}) = (\mathbf{v}_{i_0}, \ldots, \mathbf{v}_{i_{n-1}})$ is created in such a way that the entries $\mathbf{v}_{i_m}$ in $(\mathcal{V})$ are sorted according to their distance to $\mathbf{p}_{\xi_t}$ in ascending order

$$\|\mathbf{v}_{i_m} - \mathbf{p}_{\xi_t}\| \leq \|\mathbf{v}_{i_{m+1}} - \mathbf{p}_{\xi_t}\| \ , \ m \in \{0, 1, \ldots, n-2\} \ ,$$

i.e., $\mathbf{v}_{i_0}$ is the vertex closest to $\mathbf{p}_{\xi_t}$, and $\mathbf{v}_{i_{n-1}}$ is the vertex farthest away from $\mathbf{p}_{\xi_t}$ with respect to the ℓ^2-norm. Let a function k yield the rank of a vertex $\mathbf{v}_{i_k}$ in the neighborhood ranking $(\mathcal{V})$

$$k : \mathcal{V} \times \mathcal{P} \times \mathcal{V}^* \to \{0, 1, \ldots, |\mathcal{V}| - 1\} \quad ,$$

Table 2.2: Learning parameters used in the NG examples.

Parameter		Value
Step size	$\epsilon_i \dots \epsilon_f$	0.5 … 0.005
Width of lateral influence	$\lambda_i \dots \lambda_f$	10.0 … 0.01
Maximum number of iterations	$t_{\max}$	30 000

where $\mathcal{V}^*$ denotes the set of all conceivable vertex sets. Then, the SOM update rule can be rewritten using a new neighborhood function

$$h_\lambda : \mathbb{N} \times \mathbb{N} \to [0,1]$$

yielding the lateral influence of a unit according to its rank.

$\langle$*NG position updates*$\rangle \equiv$ ← pp. 30, 35

$$\mathbf{v}_i := \mathbf{v}_i - \epsilon(t) \cdot h_\lambda\big(t, k(\mathbf{v}_i, \mathbf{p}_{\xi_t}, \mathcal{V})\big) \cdot (\mathbf{v}_i - \mathbf{p}_{\xi_t}) \quad , \ \forall \mathbf{v}_i \in \mathcal{V}$$

Similar to the lateral influence of SOM Martinetz et al. used a Gaussian for h_λ that is centered at $k = 0$ with a time-dependent standard deviation $\sigma(t) = \sqrt{0.5\lambda(t)}$:

$$h_\lambda\big(t, k(\mathbf{v}_i, \mathbf{p}_{\xi_t}, \mathcal{V})\big) = \exp\left(-\frac{k(\mathbf{v}_i, \mathbf{p}_{\xi_t}, \mathcal{V})}{\lambda(t)}\right) \quad ,$$

where $\lambda(t)$ uses the same time dependence as the standard deviation $\sigma(t)$ in SOM (Eq. 2.3):

$$\lambda : \mathbb{N} \to \mathbb{R} \quad \text{where} \quad \lambda(t) = \lambda_i \cdot \left(\frac{\lambda_f}{\lambda_i}\right)^{t \cdot t_{\max}^{-1}} \quad .$$

Initially, $\lambda(t = 0) = \lambda_i$. It decreases monotonically over time $t \in \mathbb{N}$ in such a way that $\lambda(t = t_{\max}) = \lambda_f$.

Relation to k-means clustering, energy function

Similar to SOM, if $\lambda(t) \to 0$ and thus $h_\lambda\big(t, k(\mathbf{v}_i, \mathbf{p}_{\xi_t}, \mathcal{V})\big) \to \delta_{k,0}$ the Kronecker delta, then NG learning will also become equivalent to an online, stochastic variant of the k-means clustering algorithm [87, 90]. However, in contrast to SOM, NG will on average execute a stochastic gradient descent on a single energy function even if $\lambda(t) \neq 0$. Details can be found in the work of Martinetz et al. [94].

NG topology learning

After the position learning phase has been finished, the topology of the original surface is learned. For this purpose edge creation is iterated in a loop for a prespecified number $t'_{\max}$ of iterations. If a finite set of input points is used, $t'_{\max}$ should be set to the number of input points so that each point will be presented once to learn the entire topological information.

In each iteration t' a point $\mathbf{p}_{\xi_{t'}}$ is selected randomly from the set $\mathcal{P}$ of input points. For the selected $\mathbf{p}_{\xi_{t'}}$ the closest vertex $\mathbf{v}_b$ and the second-closest vertex $\mathbf{v}_c$ are determined with respect to the ℓ^2-norm.

If these vertices are not yet connected by an edge, a new edge $(\mathbf{v}_b, \mathbf{v}_c)$ connecting both will be created.

```
⟨NG topology learning⟩ ≡                                   ← p. 28
    for each $t' \in \{1, 2, \ldots, t'_{\max}\}$ do
        randomly select $\mathbf{p}_{\xi_{t'}} \in \mathcal{P}$

        $\mathbf{v}_b = \arg\min_{\mathbf{v}_i \in \mathcal{V}} \|\mathbf{v}_i - \mathbf{p}_{\xi_{t'}}\|$
        $\mathbf{v}_c = \arg\min_{\mathbf{v}_i \in \mathcal{V}\setminus\{\mathbf{v}_b\}} \|\mathbf{v}_i - \mathbf{p}_{\xi_{t'}}\|$

        $\mathcal{E} := \mathcal{E} \cup \{(\mathbf{v}_b, \mathbf{v}_c)\}$
    end for each
```

Relation to Delaunay triangulation

Martinetz and Schulten prove that the set of edges that is created during the NG topology learning phase forms a subset of the set of edges that are present in the Delaunay triangulation of the vertices in $\mathcal{V}$ [91, 93]. The proof is sketched only roughly here for the sake of a concise description. Details can be found in the original work.

Any edge connecting two vertices $\mathbf{v}_i, \mathbf{v}_j \in \mathcal{V}$ is present in the Delaunay triangulation of the vertices in $\mathcal{V}$ iff their Voronoi polyhedra* $\mathcal{D}_{\mathbf{v}_i}$ and $\mathcal{D}_{\mathbf{v}_j}$ are adjacent. For the proof Martinetz and Schulten introduce a second-order Voronoi polyhedron $\mathcal{D}_{\mathbf{v}_i,\mathbf{v}_j}$ that contains all points that are closer to both $\mathbf{v}_i$ and $\mathbf{v}_j$ than to any other vertex $\mathbf{v}_k \neq \mathbf{v}_i, \mathbf{v}_j$. This allows for a straightforward proof that two Voronoi cells $\mathcal{D}_{\mathbf{v}_i}$ and $\mathcal{D}_{\mathbf{v}_j}$ are adjacent iff the second-order Voronoi cell $\mathcal{D}_{\mathbf{v}_i,\mathbf{v}_j}$ is not empty.

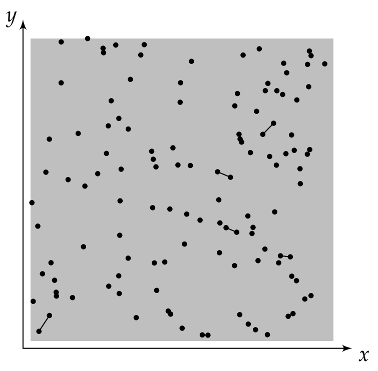
During NG topology learning $\mathcal{D}_{\mathbf{v}_b,\mathbf{v}_c}$ is obviously not empty since it contains at least $\mathbf{p}_{\xi_{t'}}$ that was sampled from the surface to be reconstructed. Thus, $\mathcal{D}_{\mathbf{v}_b}$ and $\mathcal{D}_{\mathbf{v}_c}$ are adjacent, and finally the edge $(\mathbf{v}_b, \mathbf{v}_c)$ is present in the Delaunay triangulation of the vertices. However, some $\mathcal{D}_{\mathbf{v}_i,\mathbf{v}_j}$ might not contain any selected $\mathbf{p}_{\xi_{t'}} \in \mathcal{P}$, thus NG topology learning does not create all of the edges that are present in the Delaunay triangulation. This fact is used later for motivating the development of *surface-reconstructing growing neural gas* in Sec. 4.1.

Examples for NG learning

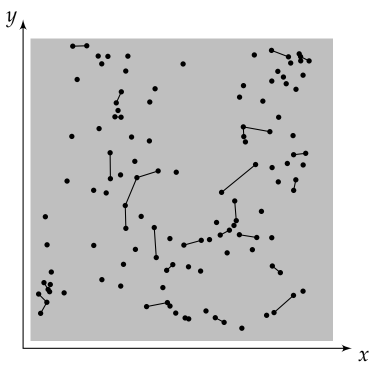
Fig. 2.8 shows an example of an initial NG configuration. The network contains 121 vertices that are placed at the position of randomly selected input points. The learning parameters that are used in the reconstruction examples are set to the values that Martinetz and Schulten [93] used (Tab. 2.2). For the figures NG position learning was interrupted after prespecified numbers $t'_{\max}$ of iterations. Then, topology learning was executed for $t'_{\max}$ iterations. Afterwards, the reconstruction was restarted from scratch.

In the first example the input points $\mathcal{P} \subset \mathcal{P}_{\square}$ (Eq. 2.1) are drawn uniformly at random from a square. Fig. 2.9 shows the resulting mesh after different numbers of iterations. It can easily be seen that the vertices tend to form clusters during the first iterations (Fig. 2.9(a)–(d)) due to a wide lateral influence. They do not collapse to a single small

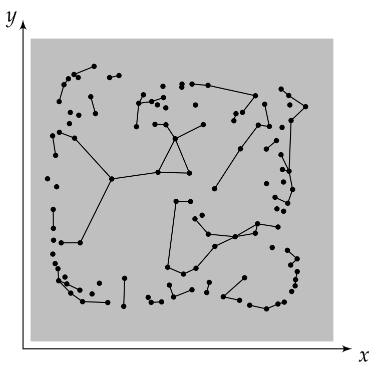
* Since Voronoi polyhedra are also called Dirichlet regions, the symbol $\mathcal{D}$ is used to avoid confusion with the set $\mathcal{V}$ of vertices.



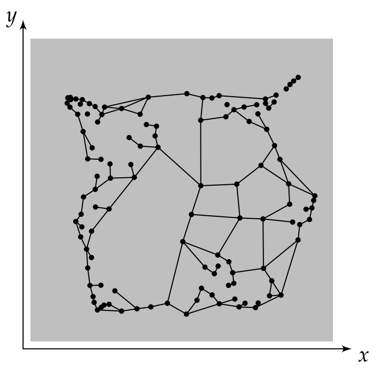
(a) $t'_{\text{max}} = 10$ iterations



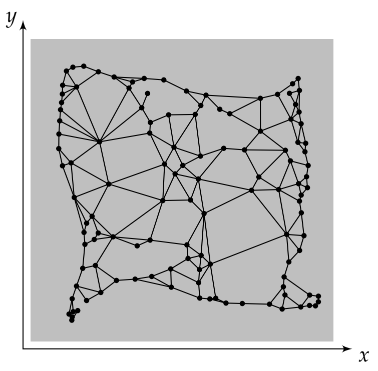
(b) $t'_{\text{max}} = 30$ iterations



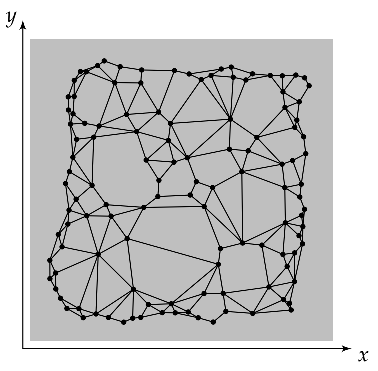
(c) $t'_{\text{max}} = 100$ iterations



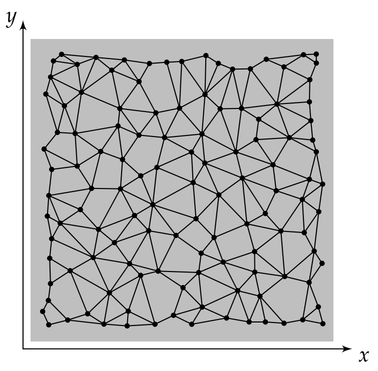
(d) $t'_{\text{max}} = 300$ iterations



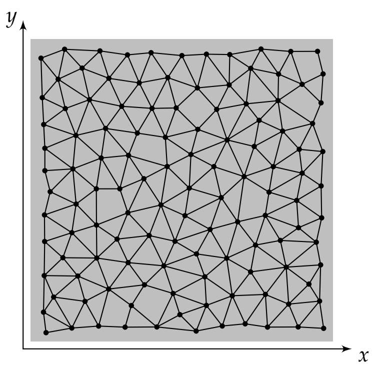
(e) $t'_{\text{max}} = 1000$ iterations



(f) $t'_{\text{max}} = 3000$ iterations

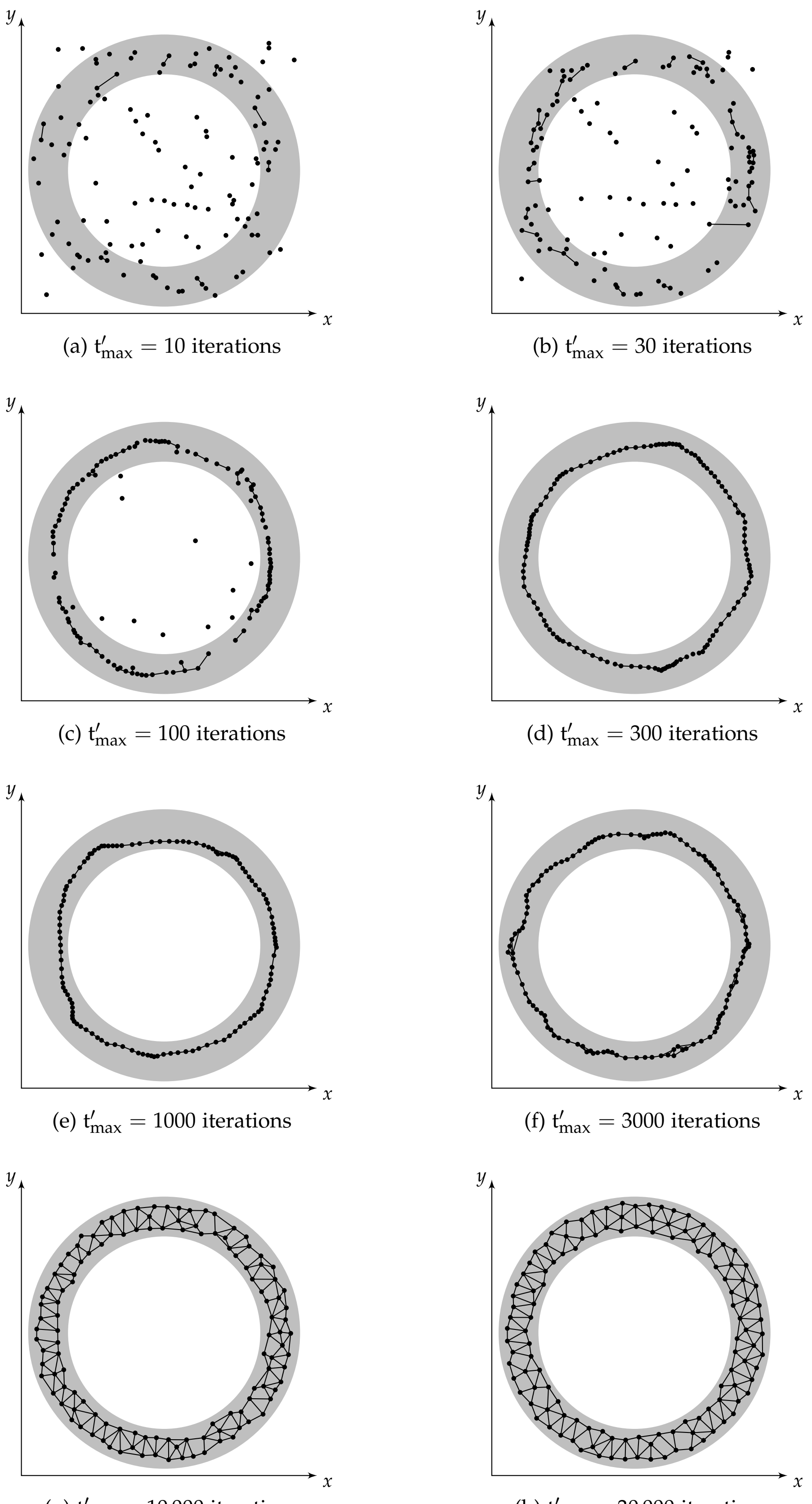


(g) $t'_{\text{max}} = 10\,000$ iterations



(h) $t'_{\text{max}} = 30\,000$ iterations

Figure 2.9: NG reconstructing a square.



(a) $t'_{max} = 10$ iterations

(b) $t'_{max} = 30$ iterations

(c) $t'_{max} = 100$ iterations

(d) $t'_{max} = 300$ iterations

(e) $t'_{max} = 1000$ iterations

(f) $t'_{max} = 3000$ iterations

(g) $t'_{max} = 10\,000$ iterations

(h) $t'_{max} = 30\,000$ iterations

Figure 2.10: NG reconstructing an annulus.

cluster as they do in SOM. While the lateral influence is narrowing down to only the best matching unit, the vertices spread out (Fig. 2.9(e)–(g)) until they are fairly evenly distributed across the input square (Fig. 2.9(h)). The more input points are selected during topology learning, the more edges are created. Eventually, the constructed mesh is nearly completely triangulated (Fig. 2.9(h)). However, some higher-order polygons cannot be triangulated by the CHL rule.

Arbitrary topologies

The topology of the surface that is reconstructed by NG is not predefined as it was for SOM. Instead, it is created during the final topology learning phase. This is illustrated in the second example where the input points $\mathcal{P} \subset \mathcal{P}_{\circledcirc}$ (Eq. 2.2) are drawn uniformly at random from an annulus. Fig. 2.10 shows the resulting mesh after different numbers of iterations. As in the previous example, the vertices form clusters initially (Fig. 2.10(a), (b)), form a ring (Fig. 2.10(c)–(e)) and eventually spread out (Fig. 2.10(f), (g)). Finally, all vertices are located on the annulus (Fig. 2.10(h)), with the hole reconstructed correctly. As before, some higher-order polygons remain untriangulated.

2.5 TOPOLOGY REPRESENTING NETWORK

NG does not simultaneously provide the complete mesh with optimized vertex positions *and* a learned topology at any time during the reconstruction process. Instead, it relies on a post-processing step to create the necessary edges. To allow for topology adjustments in each learning iteration Martinetz and Schulten [93] proposed the *topology representing network* (TRN). TRN uses the same initialization as (Sec. 2.4), and a learning algorithm that is almost identical to that of NG.

```
⟨TRN reconstruction⟩ ≡
  ⟨NG initialization  → p. 30⟩
  ⟨TRN learning  → p. 35⟩
```

TRN learning

As before, online learning is iterated in a loop for a prespecified number $t_{\max}$ of iterations. Alternatively, a predefined convergence criterion can be used, for instance, a prespecified maximal mean distance between the input points and the vertex positions.

In each iteration t of TRN learning an input point $\mathbf{p}_{\xi_t}$ is selected randomly from the set $\mathcal{P} \subset \mathbb{R}^3$ of input points, and the vertex positions are adapted according to the learning rule of NG. In contrast to NG, the selected input point is used to modify the topology of the constructed mesh in the very same iteration.

```
⟨TRN learning⟩ ≡                                    ← p. 35
  for each $t \in \{1, 2, \ldots, t_{\max}\}$ do
    randomly select $\mathbf{p}_{\xi_t} \in \mathcal{P}$

    ⟨NG position updates  → p. 31⟩
    ⟨TRN topology updates  → p. 36⟩
  end for each
```

TRN topology updates

In TRN edges are created according to the same CHL rule that was used for NG. Thus, in each iteration a new edge $(\mathbf{v}_\mathrm{b}, \mathbf{v}_\mathrm{c})$ is created, if it does not yet exist, so that the closest vertex $\mathbf{v}_\mathrm{b}$ and the second-closest vertex $\mathbf{v}_\mathrm{c}$ for $\mathbf{p}_{\xi_t}$ are connected. Upon creation, the new edge is present in the Delaunay triangulation of the vertices.

However, during TRN learning, two connected vertices might be moved to positions where they do no longer form the pair of closest and second-closest vertex for any input point. Therefore, they are no longer connected in the respective Delaunay triangulation. Thus, TRN eventually disconnects such vertices by removing obsolete edges.

⟨*TRN topology updates*⟩ ≡ ← pp. 35, 47

$$\mathbf{v}_\mathrm{b} = \arg\min_{\mathbf{v}_i \in \mathcal{V}} \|\mathbf{v}_i - \mathbf{p}_{\xi_t}\|$$
$$\mathbf{v}_\mathrm{c} = \arg\min_{\mathbf{v}_i \in \mathcal{V} \setminus \{\mathbf{v}_\mathrm{b}\}} \|\mathbf{v}_i - \mathbf{p}_{\xi_t}\|$$
$$\mathcal{E} := \mathcal{E} \cup \{(\mathbf{v}_\mathrm{b}, \mathbf{v}_\mathrm{c})\}$$

⟨*TRN obsolete edge removal* → p. 36⟩

TRN obsolete edge removal

To enable disconnecting vertices, each edge $(\mathbf{v}_k, \mathbf{v}_l) \in \mathcal{E}$ keeps track of its age a:

$$a : \mathcal{E} \mapsto \mathbb{N} \quad , \quad \text{with} \quad a\big((\mathbf{v}_k, \mathbf{v}_l)\big) := 0 \text{ for a new edge } (\mathbf{v}_k, \mathbf{v}_l) \quad .$$

The age of the edge connecting the closest vertex $\mathbf{v}_\mathrm{b}$ and the second-closest vertex $\mathbf{v}_\mathrm{c}$ is reset to zero in each iteration, whether it already existed or was just created. The age of all other edges emanating from $\mathbf{v}_\mathrm{b}$ is incremented by one. If two connected vertices $\mathbf{v}_k$, $\mathbf{v}_l$ do no longer form the pair of closest and second-closest vertex for any input point, and thus need to be disconnected, the age of the edge $(\mathbf{v}_k, \mathbf{v}_l)$ will increase since it is not reset to zero anymore. If the age exceeds a predefined threshold $\mathrm{a}_\mathrm{max}(t)$, the edge is deleted. This threshold uses the same time dependence as, e.g., the step size in SOM or NG:

$$\mathrm{a}_\mathrm{max} : \mathbb{N} \to \mathbb{R} \quad , \quad \text{where} \quad \mathrm{a}_\mathrm{max}(t) = \mathrm{a}_{\mathrm{max}_i} \cdot \left(\frac{\mathrm{a}_{\mathrm{max}_f}}{\mathrm{a}_{\mathrm{max}_i}}\right)^{t \cdot \mathrm{t}_\mathrm{max}^{-1}} \quad .$$

Initially, $\mathrm{a}_\mathrm{max}(t = 0) = \mathrm{a}_{\mathrm{max}_i}$. It decreases monotonically over time $t \in \mathbb{N}$ in such a way that $\mathrm{a}_\mathrm{max}(t = \mathrm{t}_\mathrm{max}) = \mathrm{a}_{\mathrm{max}_f}$.

⟨*TRN obsolete edge removal*⟩ ≡ ← pp. 36, 53

$$a\big((\mathbf{v}_\mathrm{b}, \mathbf{v}_\mathrm{c})\big) := 0$$
$$a\big((\mathbf{v}_\mathrm{b}, \mathbf{v}_n)\big) := a\big((\mathbf{v}_\mathrm{b}, \mathbf{v}_n)\big) + 1 \qquad , \ \forall (\mathbf{v}_\mathrm{b}, \mathbf{v}_n) \in \mathcal{E} \setminus (\mathbf{v}_\mathrm{b}, \mathbf{v}_\mathrm{c})$$
$$\mathcal{E}_{\mathrm{a}_\mathrm{max}} = \{(\mathbf{v}_k, \mathbf{v}_l) \in \mathcal{E} \mid a\big((\mathbf{v}_k, \mathbf{v}_l)\big) > \mathrm{a}_\mathrm{max}(t)\}$$
$$\mathcal{E} := \mathcal{E} \setminus \mathcal{E}_{\mathrm{a}_\mathrm{max}}$$

Examples for TRN reconstruction

For the TRN examples the same initial configuration as for NG is used with 121 vertices that are placed at the position of randomly selected input points (Fig. 2.8). The learning parameters that are used

Table 2.3: Learning parameters used in the TRN examples.

Parameter		Value
Step size	$\epsilon_i \ldots \epsilon_f$	0.5 . . . 0.005
Width of lateral influence	$\lambda_i \ldots \lambda_f$	10.0 . . . 0.01
Maximum edge age	$\mathrm{a}_{\mathrm{max},i} \ldots \mathrm{a}_{\mathrm{max},f}$	12.0 . . . 242.0
Maximum number of iterations	$\mathrm{t}_{\mathrm{max}}$	30 000

in the reconstruction examples are set to the values that Martinetz and Schulten [93] used (Tab. 2.3).

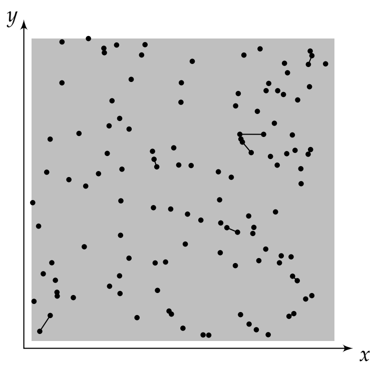
In the first example the input points $\mathcal{P} \subset \mathcal{P}_{\square}$ (Eq. 2.1) are drawn uniformly at random from a square. Fig. 2.11 shows the resulting mesh after different numbers of iterations. It can easily be seen that the vertices are located at exactly the same positions as during NG learning, since the topology updates do not affect the position updates in TRN. However, TRN creates a different set of edges. It creates more edges than NG in early iterations (Fig. 2.11(c)–(e)). Since edges are created during learning, some of them intersect. Eventually, the intersections are removed during later iterations (Fig. 2.11(f), (g)). The final mesh consists of many triangles (Fig. 2.11(h)), but more higher-order polygons remain untriangulated than in the final mesh that was constructed by NG.

In the second example the input points $\mathcal{P} \subset \mathcal{P}_{\circledcirc}$ (Eq. 2.2) are drawn uniformly at random from an annulus. Fig. 2.12 shows the resulting mesh after different numbers of iterations. As in the previous example, the vertices form clusters initially (Fig. 2.12(a), (b)), form a ring (Fig. 2.12(c)–(e)), and eventually spread out (Fig. 2.12(f), (g)). Finally, all vertices are located on the annulus (Fig. 2.12(h)), with the hole reconstructed correctly. As before, there are some higher order polygons that remain untriangulated, some more than in the example for NG.

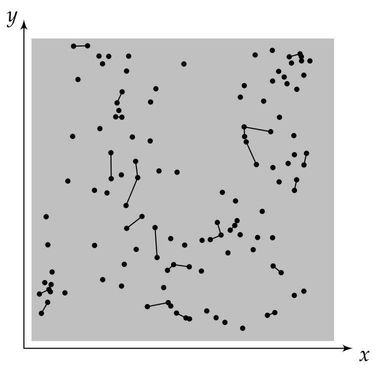
2.6 GROWING CELL STRUCTURES

Although SOM produces very good results, it is unable to create meshes of arbitrary sizes: The number of vertices is fixed and has to be predefined according to the specific task. Building upon SOM learning rules, Fritzke [49, 51, 53] proposed *growing cell structures* (GCS) in order to create meshes with an arbitrary size.

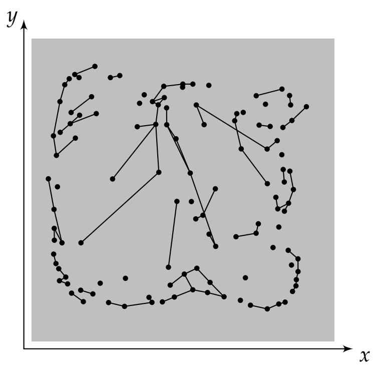
Since GCS starts with only a single k-dimensional simplex and refines it locally in a predefined way, GCS creates an ordered map by construction. That way, the lateral influence for position updates does only need to have local support without any time dependence, leading to reduced complexity and thus shorter running times. Triangles can be integrated into GCS in a straightforward way for surface reconstruction [70]. Therefore they are included in this description.



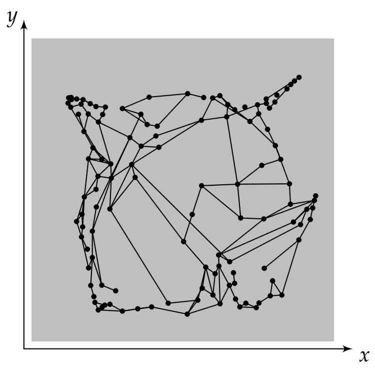
(a) $t = 10$ iterations



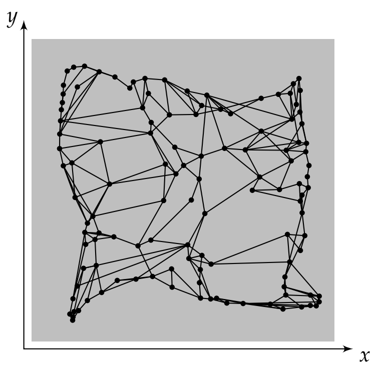
(b) $t = 30$ iterations



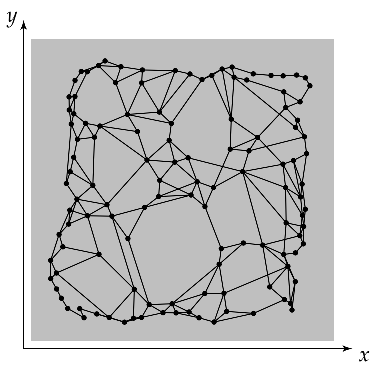
(c) $t = 100$ iterations



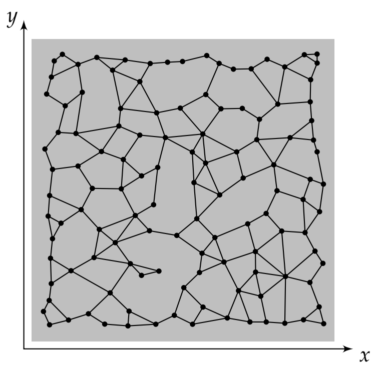
(d) $t = 300$ iterations



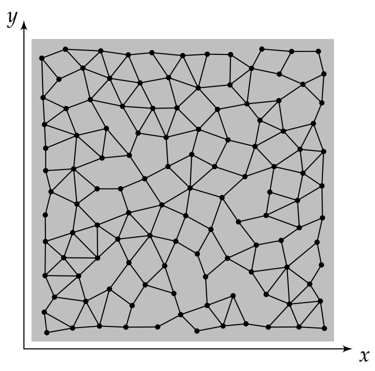
(e) $t = 1000$ iterations



(f) $t = 3000$ iterations

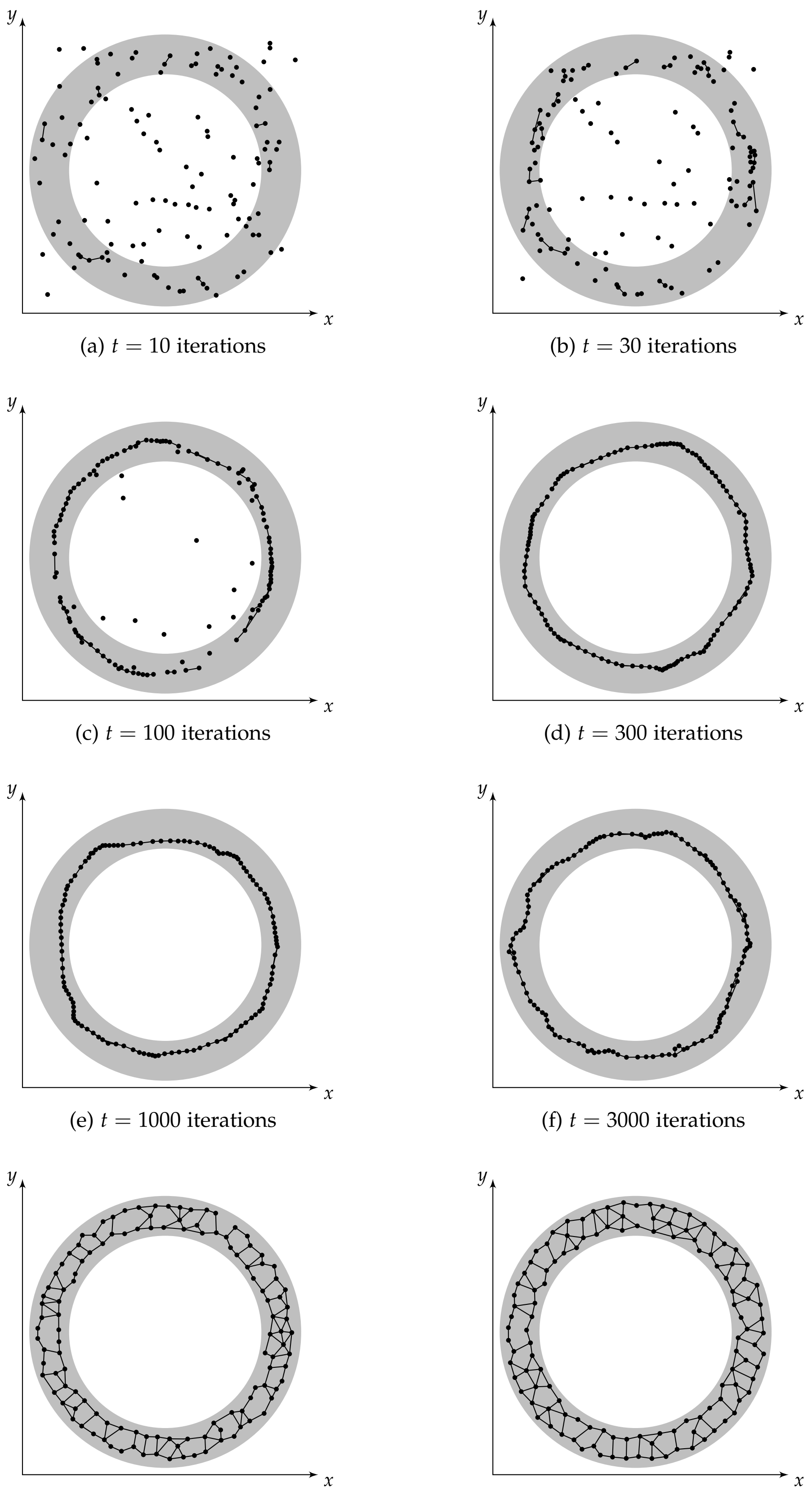


(g) $t = 10\,000$ iterations



(h) $t = 30\,000$ iterations

Figure 2.11: TRN reconstructing a square.



(a) $t = 10$ iterations

(b) $t = 30$ iterations

(c) $t = 100$ iterations

(d) $t = 300$ iterations

(e) $t = 1000$ iterations

(f) $t = 3000$ iterations

(g) $t = 10\,000$ iterations

(h) $t = 30\,000$ iterations

Figure 2.12: TRN reconstructing an annulus.

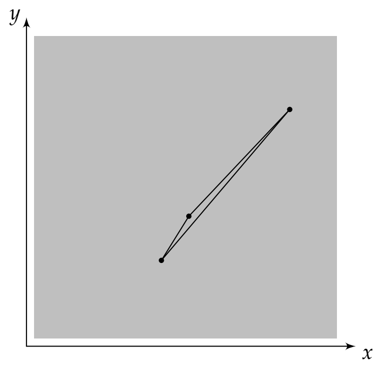


Figure 2.13: Initial configuration of GCS reconstructing a square.

The reconstruction algorithm of GCS consists of two phases.

$\langle$*GCS reconstruction*$\rangle \equiv$
  $\langle$*GCS initialization* $\twoheadrightarrow$ p. 40$\rangle$
  $\langle$*GCS learning* $\twoheadrightarrow$ p. 41$\rangle$

GCS initialization

During initialization a set $\mathcal{V}$ of $k+1$ vertices $\mathbf{v}_i \in \mathcal{V} \subset \mathbb{R}^3$ is created, with $k = 2$ if a surface has to be created that is homeomorphic to a disc, and with $k = 3$ for a surface that is homeomorphic to a sphere. The positions $\mathbf{v}_i$ of the vertices are initialized to the positions of randomly selected input points (Fig. 2.13). The vertex positions will be optimized during learning. Edges are created in such a way that all pairs of initial vertices are connected once. For surface reconstruction, triangular faces are created accordingly, so that the initial mesh forms a k-dimensional simplex, i.e., a triangle ($k = 2$) or a tetrahedron ($k = 3$).

$\langle$*GCS initialization*$\rangle \equiv$ $\twoheadleftarrow$ p. 40

$$\mathcal{V} := \{\mathbf{v}_1, \dots, \mathbf{v}_{k+1}\} \ , \ \mathbf{v}_i := \mathbf{p}_{\xi_i} \in \mathcal{P} \ ,$$
$$i \in \{1, 2, \dots, k+1\} \ , \ k \in \{2, 3\}$$
$$\mathcal{E} := \{(\mathbf{v}_i, \mathbf{v}_j) \in \mathcal{V}^2 \mid \mathbf{v}_i \neq \mathbf{v}_j\}$$
$$\mathcal{F} := \{\triangle(\mathbf{v}_l, \mathbf{v}_m, \mathbf{v}_n) \in \mathcal{V}^3 \mid (\mathbf{v}_l, \mathbf{v}_m), (\mathbf{v}_m, \mathbf{v}_n), (\mathbf{v}_n, \mathbf{v}_l) \in \mathcal{E}\}$$

GCS learning

As before, online learning is iterated in a loop for a prespecified number $\mathrm{t}_{\max}$ of iterations. Alternatively, a predefined convergence criterion can be used, for instance, a prespecified number of constructed vertices or triangles, or a prespecified maximal mean distance between the input points and the constructed mesh.

In each iteration t of GCS learning an input point $\mathbf{p}_{\xi_t}$ is selected randomly from the set $\mathcal{P} \subset \mathbb{R}^3$ of input points. For the selected input point $\mathbf{p}_{\xi_t}$ the best matching, i.e., closest, vertex $\mathbf{v}_\mathrm{b} \in \mathcal{V}$ with respect to the ℓ^2-norm $\|\cdot\|$ is determined. Afterwards, the vertex positions are adapted to $\mathbf{p}_{\xi_t}$. Finally, GCS checks whether the density of the mesh constructed so far needs to be modified in order to match the input points better, i.e., if vertices need to be added or removed.

```
⟨GCS learning⟩ ≡                                          ← p. 40
  for each $t \in \{1, 2, \ldots, t_{\max}\}$ do
      randomly select $\mathbf{p}_{\xi_t} \in \mathcal{P}$
      $\mathbf{v}_b = \arg\min_{\mathbf{v}_i \in \mathcal{V}} \|\mathbf{v}_i - \mathbf{p}_{\xi_t}\|$
      ⟨GCS position updates  → p. 41⟩
      ⟨GCS density updates  → p. 41⟩
  end for each
```

GCS position updates

Let $\mathcal{N}$ yield the set of vertices that are directly connected to another vertex by an edge

$$\mathcal{N} : \mathcal{V} \mapsto \mathbb{P}(\mathcal{V}) \quad , \text{ with } \mathcal{N}(\mathbf{v}_i) = \mathcal{N}_{\mathbf{v}_i} = \{\mathbf{v}_n \in \mathcal{V} \mid (\mathbf{v}_i, \mathbf{v}_n) \in \mathcal{E}\} \quad ,$$

where $\mathbb{P}(\cdot)$ denotes the power set. The shorthand $\mathcal{N}_{\mathbf{v}_i}$ is used instead of $\mathcal{N}(\mathbf{v}_i)$ in the remainder of this dissertation.

Since GCS creates an ordered map by construction, only the best matching vertex $\mathbf{v}_b$ that is closest to $\mathbf{p}_{\xi_t}$ and its directly connected neighbors $\mathbf{v}_n \in \mathcal{N}_{\mathbf{v}_b}$ are adapted to $\mathbf{p}_{\xi_t}$ according to update rules using constant step sizes β for $\mathbf{v}_b$, and η for its neighbors, with $\beta, \eta \in [0, 1]$. In practice, to achieve good results $\beta \gg \eta$.

```
⟨GCS position updates⟩ ≡                          ← pp. 41, 47, 52, 90
  $\mathbf{v}_b := \mathbf{v}_b - \beta(\mathbf{v}_b - \mathbf{p}_{\xi_t})$
  $\mathbf{v}_n := \mathbf{v}_n - \eta(\mathbf{v}_n - \mathbf{p}_{\xi_t})$      , $\forall \mathbf{v}_n \in \mathcal{N}_{\mathbf{v}_b}$
```

GCS density updates

In order to adapt the density of the constructed mesh to the density of the input points, each vertex $\mathbf{v}_i \in \mathcal{V}$ keeps track of its activity τ and the number ϑ of the last iteration it was selected as best match:

$$\tau : \mathcal{V} \mapsto \mathbb{R} \ , \ \tau(\mathbf{v}_i) \geq 0 \quad , \text{ with } \tau(\mathbf{v}_i) := 0 \text{ initially} \tag{2.5}$$

$$\vartheta : \mathcal{V} \mapsto \mathbb{N} \quad , \text{ with } \vartheta(\mathbf{v}_i) := t \text{ initially} \quad . \tag{2.6}$$

Whenever a vertex is selected best match for an input point, its activity is incremented by one. To weight activity in recent iterations stronger τ decays over time by a constant factor $\alpha \in [0, 1]$.

```
⟨GCS density updates⟩ ≡                                   ← p. 41
  $\tau(\mathbf{v}_b) := \tau(\mathbf{v}_b) + 1$
  $\vartheta(\mathbf{v}_b) := t$

  ⟨GCS mesh refinement  → p. 42⟩
  ⟨GCS mesh coarsening  → p. 43⟩

  $\tau(\mathbf{v}_i) := \alpha\tau(\mathbf{v}_i)$      , $\forall \mathbf{v}_i \in \mathcal{V}$
```

GCS mesh refinement

Whenever the number of iterations is an integer multiple of a predefined parameter λ_s, a new vertex $\mathbf{v}_o$ is added to the mesh. It is placed in the middle of the longest edge $(\mathbf{v}_m, \mathbf{v}_n)$ emanating from the most active vertex $\mathbf{v}_m$. In order to preserve the topology of the constructed mesh the new vertex is created by splitting the edge $(\mathbf{v}_m, \mathbf{v}_n)$ and its

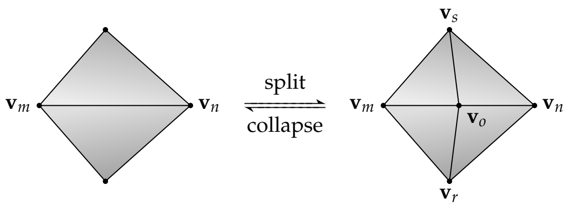


Figure 2.14: Splitting edge $(\mathbf{v}_m, \mathbf{v}_n)$, collapsing edge $(\mathbf{v}_o, \mathbf{v}_m)$.

adjacent triangles (Fig. 2.14 left to right). Activity is distributed among the existing and the new vertices according to the number $|\mathcal{N}_{\mathbf{v}_o}|$ of neighbors of the new vertex.

```
⟨GCS mesh refinement⟩ ≡                                                  ← p. 41
  if $t \equiv 0 \pmod{\lambda_s}$ then
      $\mathbf{v}_m = \arg\max_{\mathbf{v}_i \in \mathcal{V}} \tau(\mathbf{v}_i)$
      $\mathbf{v}_n = \arg\max_{\mathbf{v}_j \in \mathcal{N}_{\mathbf{v}_m}} \|\mathbf{v}_m - \mathbf{v}_j\|$
      $\mathcal{V} := \mathcal{V} \cup \{\mathbf{v}_o := \frac{\mathbf{v}_m + \mathbf{v}_n}{2}\}$
      split edge $(\mathbf{v}_m, \mathbf{v}_n)$ with new vertex $\mathbf{v}_o$
      $\tau(\mathbf{v}_k) := (1 - \frac{1}{|\mathcal{N}_{\mathbf{v}_o}|}) \cdot \tau(\mathbf{v}_k) \quad , \forall \mathbf{v}_k \in \mathcal{N}_{\mathbf{v}_o}$
      $\tau(\mathbf{v}_o) := \frac{1}{|\mathcal{N}_o|} \sum_{\mathbf{v}_k \in \mathcal{N}_o} \tau(\mathbf{v}_k)$
  end if
```

The *edge split* creates the new vertex $\mathbf{v}_o$, and it deletes the existing edge $(\mathbf{v}_m, \mathbf{v}_n)$ and the triangles adjacent to it. Afterwards, new edges are inserted so that $\mathbf{v}_o$ is connected to $\mathbf{v}_m$ and $\mathbf{v}_n$, and to their common neighbors. Finally, triangles are created for each loop of three edges that contains $\mathbf{v}_o$.

GCS mesh coarsening

Ivrissimtzis et al. [70] added a topology preserving removal of inactive vertices to GCS: Whenever the number of iterations is an integer multiple of a predefined parameter λ_c, the set $\mathcal{V}_o \subset \mathcal{V}$ of inactive vertices $\mathbf{v}_o \in \mathcal{V}_o$ is removed from the mesh. A vertex is considered inactive, if it was not selected as best matching vertex for any input point for a number of $\Delta\vartheta_{\max}$ iterations in a row. If vertices and input points are distributed uniformly, a vertex is likely selected best match in μ of $\mu \cdot |\mathcal{V}|$ iterations, where $|\mathcal{V}|$ denotes the number of vertices in the mesh. Thus, $\Delta\vartheta_{\max} = \mu \cdot |\mathcal{V}|$ yields a suitable threshold to detect inactivity robustly.

In order to preserve the topology of the constructed mesh, inactive vertices $\mathbf{v}_o \in \mathcal{V}_o$ are removed by collapsing an edge $(\mathbf{v}_o, \mathbf{v}_m)$ and its adjacent triangles (Fig. 2.14 right to left). The edge $(\mathbf{v}_o, \mathbf{v}_m)$ is selected in such a way that the topological type of the constructed mesh is preserved, and that the valences of the affected vertices are as close to

Table 2.4: Learning parameters used in the GCS examples.

Parameter		Value
Step size (best match $\mathbf{v}_\mathrm{b}$)	β	0.06
Step size (direct neighbors $\mathbf{v}_n \in \mathcal{N}_{\mathbf{v}_\mathrm{b}}$)	η	0.002
Activity decay	α	0.95
Inactivity threshold	$\Delta\vartheta_\mathrm{max}$	$6 \cdot \lvert\mathcal{V}\rvert$
Iterations for split and collapse	$\lambda_\mathrm{s}, \lambda_\mathrm{c}$	100

six as possible. Hoppe et al. [65] defined criteria for the former, and Ivrissimtzis et al. [70] introduced a regularity error E_c for the latter:

$$E_c : \mathcal{E} \to \mathbb{R} \quad , \quad E_c\big((\mathbf{v}_o, \mathbf{v}_m)\big) \geq 0 \quad ,$$

$$\text{with} \quad E_c\big((\mathbf{v}_o, \mathbf{v}_m)\big) = \\ (|\mathcal{N}_{\mathbf{v}_m}| + |\mathcal{N}_{\mathbf{v}_o}| - 10)^2 + (|\mathcal{N}_{\mathbf{v}_r}| - 7)^2 + (|\mathcal{N}_{\mathbf{v}_s}| - 7)^2 \quad , \tag{2.7}$$

where $|\mathcal{N}_{\mathbf{v}_k}|$ denotes the number of neighbors of a vertex $\mathbf{v}_k$, and $\mathbf{v}_o$ denotes the inactive vertex, $\mathbf{v}_m$ denotes the other endpoint of the selected edge, and $\mathbf{v}_r$, $\mathbf{v}_s$ denote the two affected neighbors (Fig. 2.14 right). GCS selects the $\mathbf{v}_m$ that causes the lowest $E_c\big((\mathbf{v}_o, \mathbf{v}_m)\big)$.

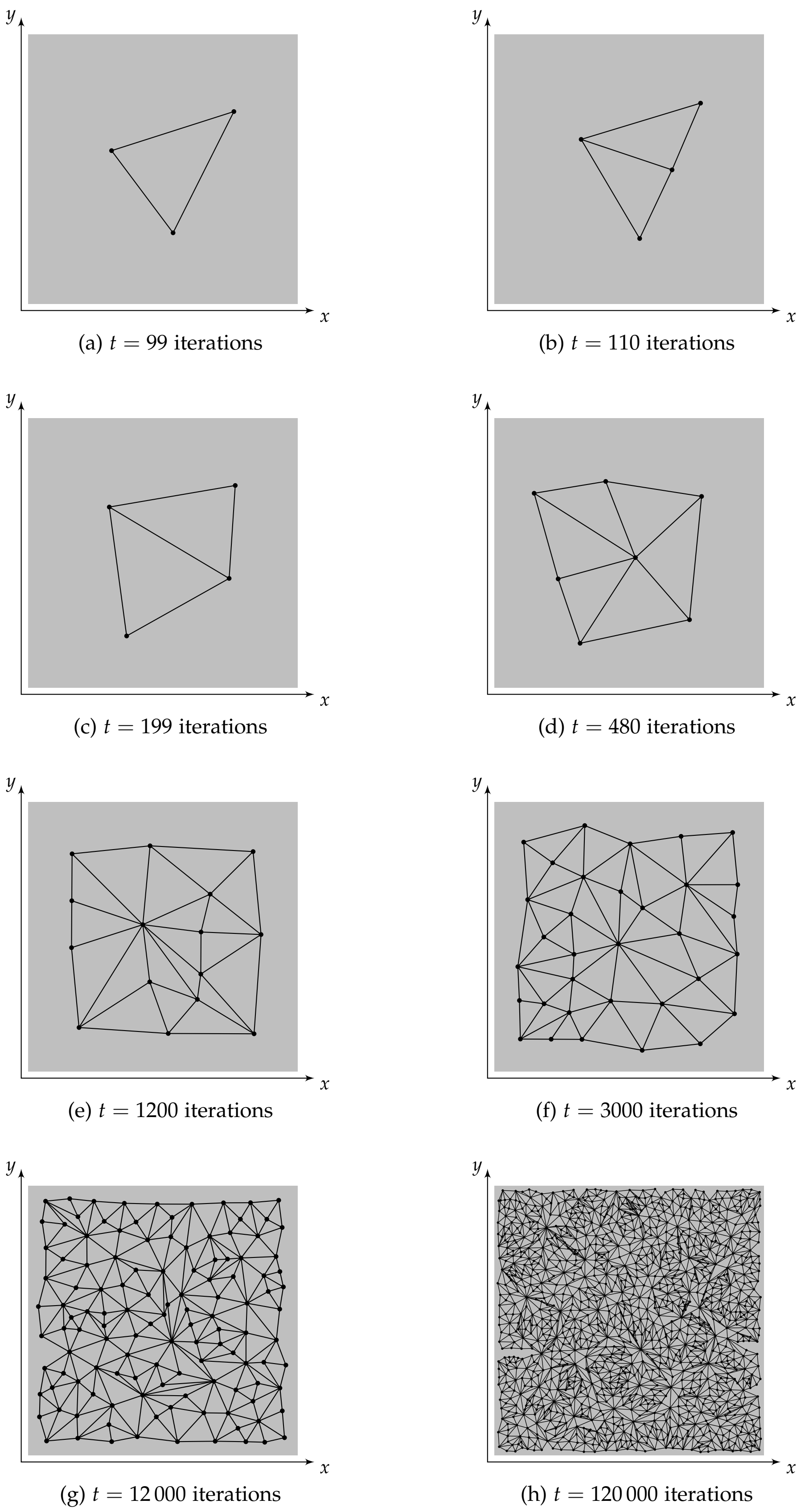
```
⟨Gcs mesh coarsening⟩ ≡                                              ← p. 41
  if $t \equiv 0 \pmod{\lambda_\mathrm{c}}$ then
    $\mathcal{V}_o = \{\mathbf{v}_o \in \mathcal{V} \mid \vartheta(\mathbf{v}_o) < t - \Delta\vartheta_\mathrm{max}\}$
    for each $\mathbf{v}_o \in \mathcal{V}_o$ do
      $\mathcal{V}_o^+ = \{\mathbf{v}_l \in \mathcal{N}_{\mathbf{v}_o} \mid$ collapsing $(\mathbf{v}_o, \mathbf{v}_l)$ is valid [65]$\}$
      if $|\mathcal{V}_o^+| > 0$ then
        $\mathbf{v}_m = \arg\min_{\mathbf{v}_l \in \mathcal{V}_o^+} E_c\big((\mathbf{v}_o, \mathbf{v}_l)\big)$
        collapse edge $(\mathbf{v}_o, \mathbf{v}_m)$ removing vertex $\mathbf{v}_o$
      end if
    end for each
  end if
```

An *edge collapse* first deletes the edge $(\mathbf{v}_o, \mathbf{v}_m)$ and its adjacent triangles. Additionally, the edges connecting $\mathbf{v}_o$ to the common neighbors of $\mathbf{v}_m$ and $\mathbf{v}_o$ are deleted, but their adjacent triangles are preserved and attached to $\mathbf{v}_m$. Afterwards, the remaining edges emanating from $\mathbf{v}_o$ and their adjacent triangles are detached from $\mathbf{v}_o$ and attached to $\mathbf{v}_m$. Finally, the inactive vertex $\mathbf{v}_o$ is deleted.

Examples for GCS learning

Fig. 2.13 shows an example of an initial GCS configuration with $k = 2$. The network initially contains three vertices that are placed at the positions of three randomly selected input points. The learning parameters that are used in the reconstruction examples are set to the values that Fritzke [51] used except for $\Delta\vartheta_\mathrm{max}$ (Tab. 2.4).

In the first example the input points $\mathcal{P} \subset \mathcal{P}_\square$ (Eq. 2.1) are drawn uniformly at random from a square. Fig. 2.15 shows the resulting mesh after different numbers of iterations. During the first 99 iterations the



(a) $t = 99$ iterations

(b) $t = 110$ iterations

(c) $t = 199$ iterations

(d) $t = 480$ iterations

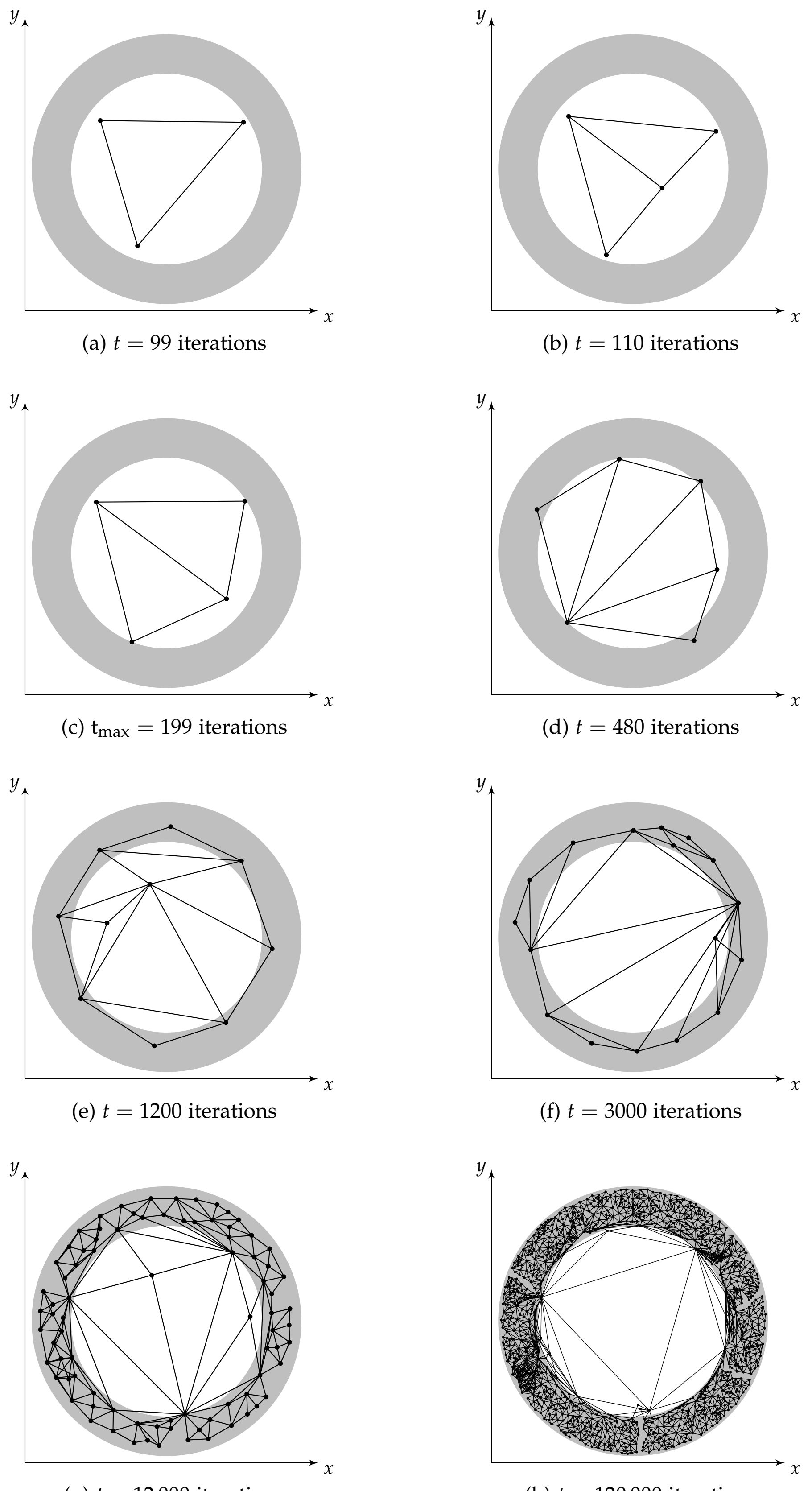
(e) $t = 1200$ iterations

(f) $t = 3000$ iterations

(g) $t = 12\,000$ iterations

(h) $t = 120\,000$ iterations

Figure 2.15: GCS reconstructing a square.



(a) $t = 99$ iterations

(b) $t = 110$ iterations

(c) $t_{\max} = 199$ iterations

(d) $t = 480$ iterations

(e) $t = 1200$ iterations

(f) $t = 3000$ iterations

(g) $t = 12\,000$ iterations

(h) $t = 120\,000$ iterations

Figure 2.16: Gcs reconstructing an annulus.

initial triangle grows to match the input points better (Fig. 2.15(a)). It is split once $t \geq \lambda_s$ (Fig. 2.15(b)). Afterwards, the triangles grow again (Fig. 2.15(c)). This process is repeated during the subsequent iterations (Fig. 2.15(d)–(g)) until the refined mesh covers the square almost completely (Fig. 2.15(h)). The constructed triangles have very different sizes. Quite a few are elongated, some are even close to degenerate. Furthermore, several fold-overs occur.

In the second example the input points $\mathcal{P} \subset \mathcal{P}_{\circledcirc}$ (Eq. 2.2) are drawn uniformly at random from an annulus. Fig. 2.16 shows the resulting mesh after different numbers of iterations. The initial triangle grows and is split with the process repeating as in the first example (Fig. 2.16(a)–(g)). Eventually, all of the constructed vertices are located on or close to the annulus (Fig. 2.16(h)). The constructed triangles have very different sizes. Quite a few are elongated, some are even close to degenerate. It can easily be seen that GCS is unable to reconstruct the topology of the input data: The learning algorithm constructs a mesh that bridges the hole in the center of the annulus. Furthermore, there are many fold-overs and some gaps.

2.7 GROWING NEURAL GAS

SOM, NG, TRN, and GCS are suitable for surface reconstruction, but they are rather inflexible: For SOM, NG, and TRN the number of vertices in the constructed mesh needs to be prespecified and remains fixed during learning. For SOM and GCS the topology of the surface needs to be prespecified and remains fixed. However, NG and TRN learn the topology, and GCS learns the number of vertices.

In order to construct meshes with arbitrary topology *and* an arbitrary number of vertices Fritzke introduced *growing neural gas* (GNG) [52, 53] which combines the capabilities of TRN and GCS. Similar to GCS, GNG creates an ordered map by construction. That way, the lateral influence for position updates does only need to have local support without any time dependence, leading to reduced complexity and thus shorter running times compared to TRN.

The reconstruction algorithm of GNG is a combination of the algorithms used in TRN and GCS. It consists of two phases.

$\langle$*GNG reconstruction*$\rangle \equiv$
  $\langle$*GNG initialization* ➝ p. 47$\rangle$
  $\langle$*GNG learning* ➝ p. 47$\rangle$

GNG initialization

During initialization a set $\mathcal{V}$ of two vertices $\mathbf{v}_i \in \mathcal{V} \subset \mathbb{R}^3$ is created. The positions $\mathbf{v}_i$ of the vertices are initialized to the positions of two randomly selected input points (Fig. 2.17). The vertex positions will be optimized during learning. No edges are created as in NG and TRN, since they are created during learning.

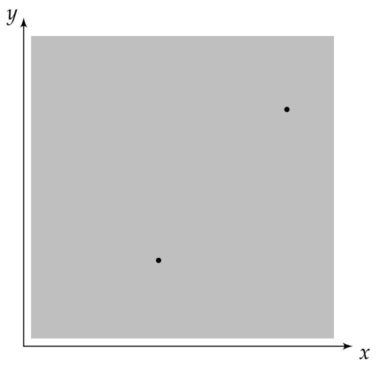


Figure 2.17: Initial configuration of GNG reconstructing a square.

```
⟨GNG initialization⟩ ≡                                    ← p. 46
  $\mathcal{V} := \{\mathbf{v}_1, \mathbf{v}_2\}$ ,  $\mathbf{v}_i := \mathbf{p}_{\xi_i} \in \mathcal{P}$ ,  $i \in \{1,2\}$
  $\mathcal{E} := \emptyset$
```

GNG learning

As before, online learning is iterated in a loop for a prespecified number t_{max} of iterations. Alternatively, a predefined convergence criterion can be used, for instance, a prespecified number of constructed vertices, or a prespecified maximal mean distance between the input points and the vertex positions.

In each iteration t of GNG learning an input point $\mathbf{p}_{\xi_t}$ is selected randomly from the set $\mathcal{P} \subset \mathbb{R}^3$ of input points. For the selected input point $\mathbf{p}_{\xi_t}$ the best matching, i.e., closest, vertex $\mathbf{v}_b \in \mathcal{V}$, and the second-best matching vertex $\mathbf{v}_c \in \mathcal{V}$ with respect to the ℓ^2-norm $\|\cdot\|$ are determined. In order to adapt the density of the constructed mesh to the density of the input points, each vertex $\mathbf{v}_i \in \mathcal{V}$ keeps track of its activity τ (Eq. 2.5) that decays over time like in GCS. The activity of the best matching vertex is incremented by its squared distance to the input point. Afterwards, the vertex positions are updated according to the position update rules of GCS, the topology is updated according to the topology update rules of TRN, and the density is updated in a similar way as in GCS.

```
⟨GNG learning⟩ ≡                                          ← p. 46
  for each $t \in \{1,2,\dots,t_{max}\}$ do
      randomly select $\mathbf{p}_{\xi_t} \in \mathcal{P}$

      $\mathbf{v}_b = \arg\min_{\mathbf{v}_i \in \mathcal{V}} \|\mathbf{v}_i - \mathbf{p}_{\xi_t}\|$
      $\mathbf{v}_c = \arg\min_{\mathbf{v}_i \in \mathcal{V} \setminus \{\mathbf{v}_b\}} \|\mathbf{v}_i - \mathbf{p}_{\xi_t}\|$

      $\tau(\mathbf{v}_b) := \tau(\mathbf{v}_b) + \|\mathbf{v}_b - \mathbf{p}_{\xi_t}\|^2$

      ⟨GCS position updates  → p. 41⟩
      ⟨TRN topology updates  → p. 36⟩
      ⟨GNG density updates  → p. 48⟩

      $\tau(\mathbf{v}_i) := \alpha\tau(\mathbf{v}_i)$   ,  $\forall \mathbf{v}_i \in \mathcal{V}$
  end for each
```

Table 2.5: Learning parameters used in the GNG examples.

Parameter		Value
Step size (best match $\mathbf{v}_\text{b}$)	β	0.2
Step size (direct neighbors $\mathbf{v}_n \in \mathcal{N}_{\mathbf{v}_\text{b}}$)	η	0.006
Maximum edge age	a_max	50
Activity decay	α	0.995
Iterations until mesh refinement	λ_s	100
Activity factor on mesh refinement	α_s	0.5

GNG density updates

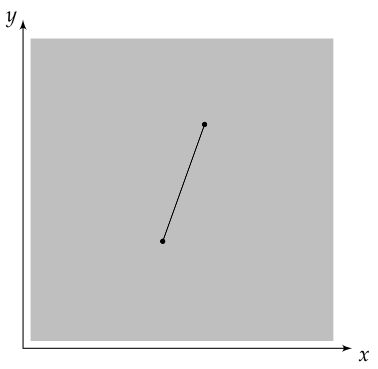
Whenever the number of iterations is an integer multiple of a predefined parameter λ_s, a new vertex $\mathbf{v}_o$ is added to the constructed mesh. It is placed in the middle of the edge $(\mathbf{v}_m, \mathbf{v}_n)$ connecting the most active vertex $\mathbf{v}_m$ to its most active neighbor $\mathbf{v}_n$. Afterwards, $\mathbf{v}_o$ is connected to $\mathbf{v}_m$ and $\mathbf{v}_n$, whereas the original edge $(\mathbf{v}_m, \mathbf{v}_n)$ is deleted. Finally, the activity of $\mathbf{v}_m$ and $\mathbf{v}_n$ is decreased by a prespecified factor α_s. The activity of the new vertex is initialized with the activity of $\mathbf{v}_m$.

```
⟨GNG density updates⟩ ≡                                    ← pp. 47, 53
  if $t \equiv 0 \pmod{\lambda_\text{s}}$ then
      $\mathbf{v}_m = \arg\max_{\mathbf{v}_i \in \mathcal{V}} \tau(\mathbf{v}_i)$
      $\mathbf{v}_n = \arg\max_{\mathbf{v}_i \in \mathcal{N}_{\mathbf{v}_m}} \tau(\mathbf{v}_i)$
      $\mathcal{V} := \mathcal{V} \cup \{\mathbf{v}_o := \frac{\mathbf{v}_m + \mathbf{v}_n}{2}\}$
      $\mathcal{E} := (\mathcal{E} \cup \{(\mathbf{v}_m, \mathbf{v}_o), (\mathbf{v}_o, \mathbf{v}_n)\}) \setminus (\mathbf{v}_m, \mathbf{v}_n)$
      $\tau(\mathbf{v}_m) := \alpha_\text{s}\tau(\mathbf{v}_m)$
      $\tau(\mathbf{v}_n) := \alpha_\text{s}\tau(\mathbf{v}_n)$
      $\tau(\mathbf{v}_o) := \tau(\mathbf{v}_m)$
  end if
```

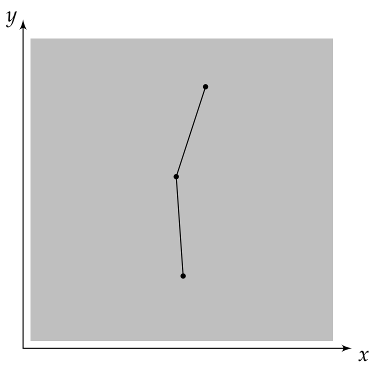
Examples for GNG learning

Fig. 2.17 shows an example of an initial GNG configuration. The network initially contains two vertices that are placed at the position of two randomly selected input points. The learning parameters that are used in the reconstruction examples are set to the values that Fritzke [52] used (Tab. 2.5).

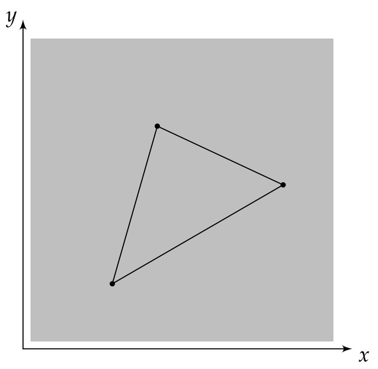
In the first example the input points $\mathcal{P} \subset \mathcal{P}_\square$ (Eq. 2.1) are drawn uniformly at random from a square. Fig. 2.18 shows the resulting mesh after different numbers of iterations. In the first iteration both initial vertices get connected by an edge. During the first 99 iterations this line segment moves and gets longer so that the vertices match the input points better (Fig. 2.18(a)). It is split and it bends once $t \geq \lambda_s$ (Fig. 2.18(b)). The vertices get connected to a first triangle during the next few iterations (Fig. 2.18(c)). Afterwards, the constructed mesh is refined repeatedly. The vertices spread out, new edges are added and obsolete ones are deleted (Fig. 2.18(d)–(g)). Eventually, the constructed mesh covers the input square almost completely (Fig. 2.18(h)). Some intersecting edges exist. Many triangles have been created with only



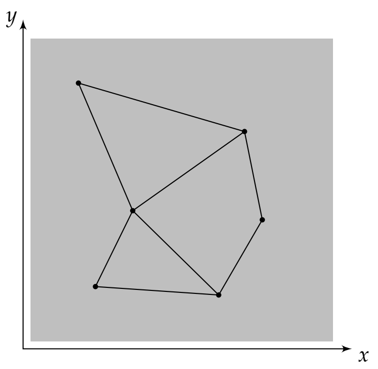
(a) $t = 99$ iterations



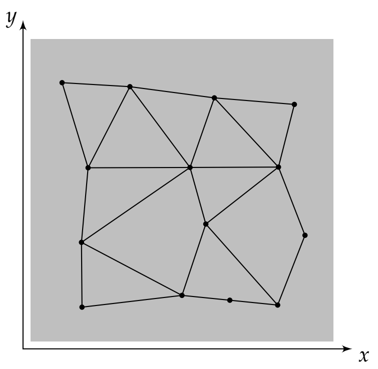
(b) $t = 110$ iterations



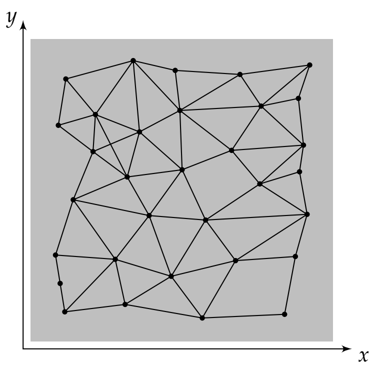
(c) $t = 199$ iterations



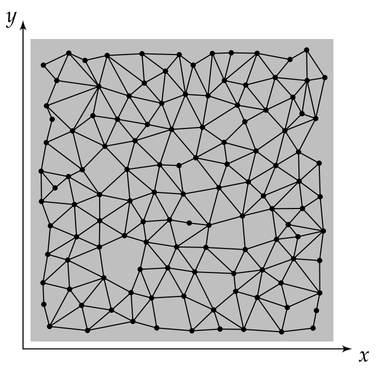
(d) $t = 480$ iterations



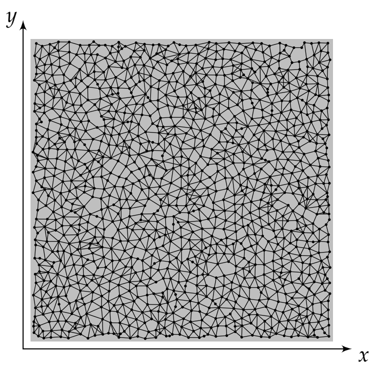
(e) $t = 1200$ iterations



(f) $t = 3000$ iterations

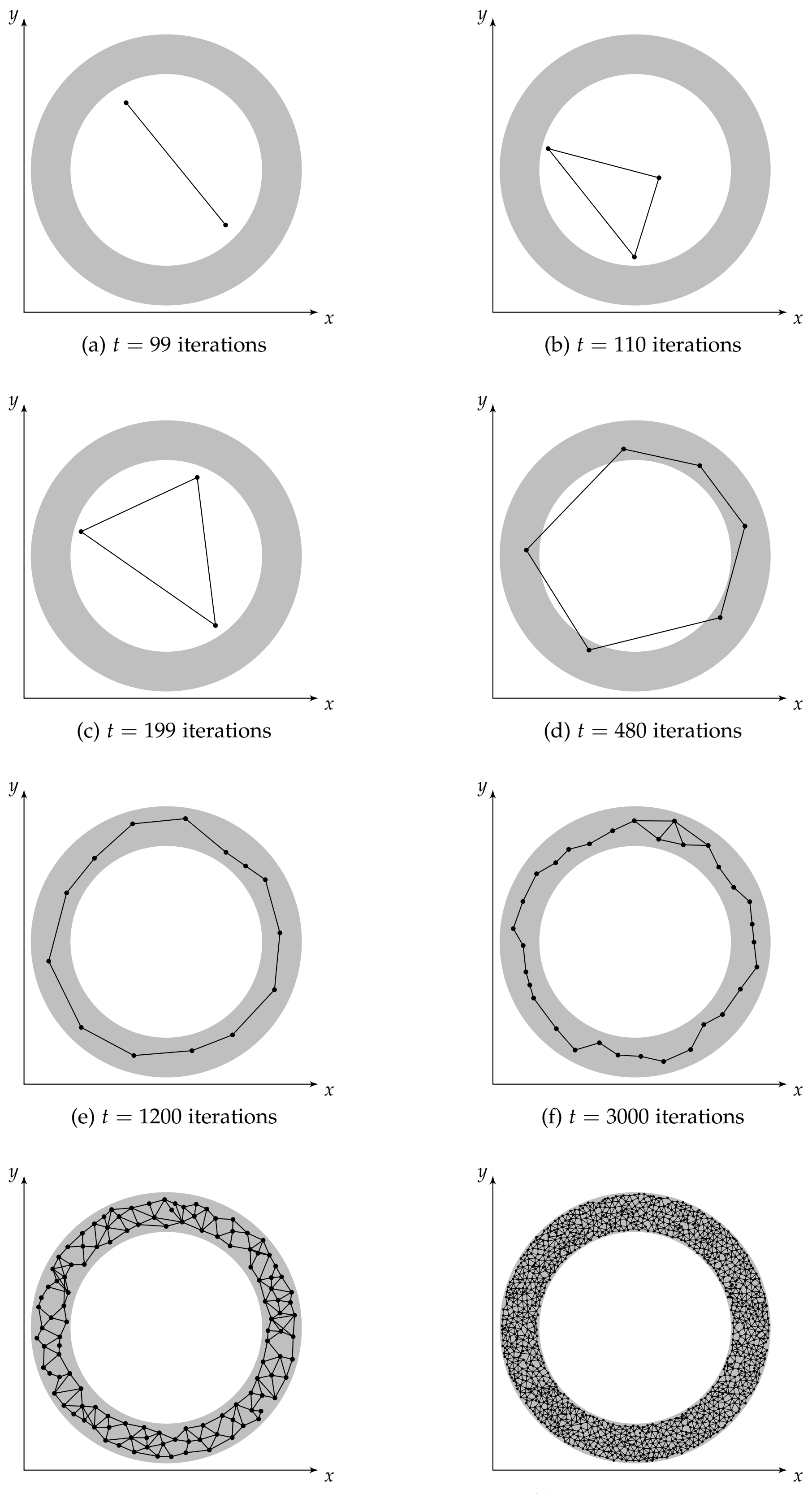


(g) $t = 12\,000$ iterations



(h) $t = 120\,000$ iterations

Figure 2.18: GNG reconstructing a square.



(a) $t = 99$ iterations

(b) $t = 110$ iterations

(c) $t = 199$ iterations

(d) $t = 480$ iterations

(e) $t = 1200$ iterations

(f) $t = 3000$ iterations

(g) $t = 12\,000$ iterations

(h) $t = 120\,000$ iterations

Figure 2.19: GNG reconstructing an annulus.

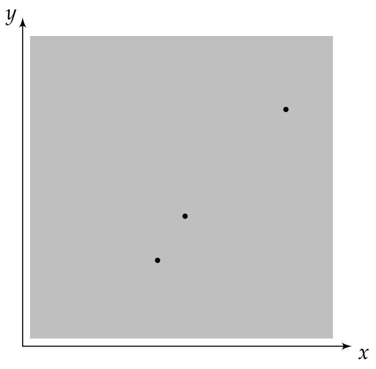


Figure 2.20: Initial configuration of GSRM reconstructing a square.

very few being degenerate. However, several higher order polygons remain untriangulated. Only few edges intersect. The triangulation created by GNG (Fig. 2.18(h)) is apparently much more regular than the one created by GCS (Fig. 2.15(h)).

In the second example the input points $\mathcal{P} \subset \mathcal{P}_{\circledcirc}$ (Eq. 2.2) are drawn uniformly at random from an annulus. Fig. 2.19 shows the resulting mesh after different numbers of iterations. The characteristics of the results are very similar to the square example. However, at first a ring of connected vertices is created (Fig. 2.19(a)–(e)). Eventually, the vertices spread out (Fig. 2.19(f), (g)). Finally, all constructed vertices are located on the input annulus (Fig. 2.19(h)). As before, some intersecting edges exist, and many triangles have been created with only very few being degenerate, but several higher-order polygons remain untriangulated.

2.8 growing self-reconstruction map

GNG combines the quality of TRN with the growing mechanism of GCS, but does not create triangular faces during learning. To overcome this—at least to some extent—do Rêgo et al. [41] proposed the *growing self-reconstruction map* (GSRM) that extends GNG by creating some triangles during learning and the remaining ones during a separate post-processing step, similar to the one used in NG (Sec. 2.4).

The basic reconstruction algorithm of GSRM consists of three phases.

⟨*GSRM reconstruction*⟩ ≡
  ⟨*GSRM initialization* → p. 52⟩
  ⟨*GSRM learning* → p. 52⟩
  ⟨*GSRM post-processing* → p. 54⟩

GSRM initialization

GSRM is initialized in a similar way as GNG. During initialization a set $\mathcal{V}$ of three vertices $\mathbf{v}_i \in \mathcal{V} \subset \mathbb{R}^3$ is created, instead of two as in GNG. The positions $\mathbf{v}_i$ of the vertices are initialized to the positions of three randomly selected input points (Fig. 2.20). No edges are created as in NG, TRN, and GNG, since they are created during learning. Finally, an initially empty set of triangles is created.

```
⟨GSRM initialization⟩ ≡                                    ← p. 51
    $\mathcal{V} := \{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3\}$ ,  $\mathbf{v}_i := \mathbf{p}_{\xi_i} \in \mathcal{P}$ ,  $i \in \{1,2,3\}$
    $\mathcal{E} := \emptyset$
    $\mathcal{F} := \emptyset$
```

GSRM learning GSRM uses a learning algorithm that resembles the one of GNG. However, GSRM topology updates create triangles and use extended edge creation and removal schemes. For GSRM notation is extended: Let $\mathcal{F}_e(\cdot)$ yield the set of faces that are adjacent to an edge:

$$\mathcal{F}_e : \mathcal{E} \to \mathbb{P}(\mathcal{F}) \quad ,$$
$$\text{with} \quad \mathcal{F}_e\big((\mathbf{v}_i, \mathbf{v}_j)\big) = \{\triangle(\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k) \in \mathcal{F} \mid \mathbf{v}_k \in \mathcal{N}_{\mathbf{v}_i} \cap \mathcal{N}_{\mathbf{v}_j}\} \quad ,$$

where $\mathbb{P}(\cdot)$ denotes the power set.

As before, online learning is iterated in a loop for a prespecified number $t_{\max}$ of iterations. Alternatively, a predefined convergence criterion can be used, for instance, a prespecified number of constructed vertices or triangles, or a prespecified maximal mean distance between the input points and the constructed mesh.

In each iteration t of GSRM learning an input point $\mathbf{p}_{\xi_t}$ is selected randomly from the set $\mathcal{P} \subset \mathbb{R}^3$ of input points. For the selected input point $\mathbf{p}_{\xi_t}$ the best matching, i.e., closest, vertex $\mathbf{v}_b \in \mathcal{V}$, and the second-best matching vertex $\mathbf{v}_c \in \mathcal{V}$ with respect to the ℓ^2-norm $\|\cdot\|$ are determined. In order to update the density of the constructed mesh according to the density of the input points, each vertex $\mathbf{v}_i \in \mathcal{V}$ keeps track of its activity τ (Eq. 2.5) that decays over time. The activity of the best matching vertex is incremented by its squared distance to the input point. Afterwards, the vertex positions are updated according to the position update rules of GCS. Then, the topology is updated according to a more elaborate rule than in TRN and GNG. Finally, the density is updated as in GNG, with the extensions that the adjacent faces are deleted whenever an edge is deleted, and that the activity of the new vertex is interpolated*.

```
⟨GSRM learning⟩ ≡                                          ← p. 51
    for each $t \in \{1, 2, \ldots, t_{\max}\}$ do
        randomly select $\mathbf{p}_{\xi_t} \in \mathcal{P}$

        $\mathbf{v}_b = \arg\min_{\mathbf{v}_i \in \mathcal{V}} \|\mathbf{v}_i - \mathbf{p}_{\xi_t}\|$
        $\mathbf{v}_c = \arg\min_{\mathbf{v}_i \in \mathcal{V} \setminus \{\mathbf{v}_b\}} \|\mathbf{v}_i - \mathbf{p}_{\xi_t}\|$

        $\tau(\mathbf{v}_b) := \tau(\mathbf{v}_b) + \|\mathbf{v}_b - \mathbf{p}_{\xi_t}\|^2$

        ⟨GCS position updates → p. 41⟩
        ⟨GSRM topology and surface updates → p. 53⟩
```

* Thus, add $\mathcal{F} := \mathcal{F} \setminus \mathcal{F}_e\big((\mathbf{v}_m, \mathbf{v}_n)\big)$ and $\tau(\mathbf{v}_o) := \frac{\tau(\mathbf{v}_m) + \tau(\mathbf{v}_n)}{2}$.

```
    ⟨GNG density updates → p. 48⟩*

    $\tau(\mathbf{v}_i) := \alpha\tau(\mathbf{v}_i) \quad , \forall \mathbf{v}_i \in \mathcal{V}$
end for each
```

GSRM topology and surface updates

In GSRM edges are created similar to the original CHL rule that was already used in TRN and in GNG. However, do Rêgo et al. [41] extended CHL in order to make it better suited for surface reconstruction: In each iteration a new edge $(\mathbf{v}_b, \mathbf{v}_c)$ is created if it does not yet exist, and—in addition to the original CHL rule—if the best $\mathbf{v}_b$ and the second-best matching vertex $\mathbf{v}_c$ do not share more than two common neighbors. If they shared three or more neighbors and were connected by a new edge, then three or more faces would be created adjacent to that edge, violating the desired 2-manifold property of the constructed mesh.

Furthermore, if $\mathbf{v}_b$ and $\mathbf{v}_c$ share two common neighbors $\mathbf{v}_i$, $\mathbf{v}_j$ that are connected, then the connecting edge $(\mathbf{v}_i, \mathbf{v}_j)$ and its adjacent triangles will be removed before creating the new edge $(\mathbf{v}_b, \mathbf{v}_c)$. That way, overlapping or intersecting edges are avoided.

If the new edge $(\mathbf{v}_b, \mathbf{v}_c)$ generates one or two loops of three edges, and if each of the edges in the respective loop has less than two adjacent triangles, then a new triangle will be created for the respective loop. In order to improve the quality of the constructed mesh, obtuse triangles are removed. Finally, obsolete edges are detected and removed by aging as in TRN and GNG, with the extension that the adjacent faces are deleted whenever an edge is deleted†.

```
⟨GSRM topology and surface updates⟩ ≡                    ← pp. 52, 54
  if $\mathcal{N}_{\mathbf{v}_b} \cap \mathcal{N}_{\mathbf{v}_c} = \{\mathbf{v}_i, \mathbf{v}_j\} \wedge (\mathbf{v}_i, \mathbf{v}_j) \in \mathcal{E}$ then
      $\mathcal{F} := \mathcal{F} \setminus \mathcal{F}_e\big((\mathbf{v}_i, \mathbf{v}_j)\big)$
      $\mathcal{E} := \mathcal{E} \setminus \{(\mathbf{v}_i, \mathbf{v}_j)\}$
  end if

  if $|\mathcal{N}_{\mathbf{v}_b} \cap \mathcal{N}_{\mathbf{v}_c}| \leq 2 \wedge (\mathbf{v}_b, \mathbf{v}_c) \notin \mathcal{E}$ then
      $\mathcal{E} := \mathcal{E} \cup \{(\mathbf{v}_b, \mathbf{v}_c)\}$
      $\mathcal{F} := \mathcal{F} \cup \{\triangle(\mathbf{v}_b, \mathbf{v}_c, \mathbf{v}_n) \mid \mathbf{v}_n \in \mathcal{N}_{\mathbf{v}_b} \cap \mathcal{N}_{\mathbf{v}_c} \ ,$
                                        $|\mathcal{F}_e\big((\mathbf{v}_c, \mathbf{v}_n)\big)| < 2 \ ,$
                                          $|\mathcal{F}_e\big((\mathbf{v}_n, \mathbf{v}_b)\big)| < 2\}$
  else if $(\mathbf{v}_b, \mathbf{v}_c) \in \mathcal{E}$ then
      ⟨GSRM obtuse triangle removal → p. 54⟩
  end if
  ⟨TRN obsolete edge removal → p. 36⟩†
```

GSRM obtuse triangle removal

Whenever the edge $(\mathbf{v}_b, \mathbf{v}_c)$ already existed, then for each edge $(\mathbf{v}_b, \mathbf{v}_i)$ emanating from $\mathbf{v}_b$ the Thales circle in the plane that contains $\mathbf{v}_b$, $\mathbf{v}_c$, and $\mathbf{v}_i$ is determined in order to detect and remove obtuse triangles. If $\mathbf{v}_c$ lies inside that circle, the triangle $\triangle(\mathbf{v}_b, \mathbf{v}_i, \mathbf{v}_c)$ is obtuse,

* Thus, add $\mathcal{F} := \mathcal{F} \setminus \mathcal{F}_e\big((\mathbf{v}_m, \mathbf{v}_n)\big)$ and $\tau(\mathbf{v}_o) := \frac{\tau(\mathbf{v}_m)+\tau(\mathbf{v}_n)}{2}$.

† Thus, add $\mathcal{F} := \mathcal{F} \setminus \mathcal{F}_e\big((\mathbf{v}_k, \mathbf{v}_l)\big)$, $\forall (\mathbf{v}_k, \mathbf{v}_l) \in \mathcal{E}_{a_{\max}}$.

Table 2.6: Learning parameters used in the GSRM examples.

Parameter		Value
Step size (best match $\mathbf{v}_b$)	β	0.05
Step size (direct neighbors $\mathbf{v}_n \in \mathcal{N}_{\mathbf{v}_b}$)	η	0.0006
Maximum edge age	a_{max}	30
Activity decay	α	0.9995
Iterations until mesh refinement	λ_s	200
Activity factor on mesh refinement	α_s	0.5

i.e., one of its angles is greater than 90°. Thus, edge $(\mathbf{v}_b, \mathbf{v}_i)$ and its adjacent triangles are deleted.

⟨*GSRM obtuse triangle removal*⟩ ≡ ← p. 53

$$\mathcal{E}_o = \{(\mathbf{v}_b, \mathbf{v}_i) \in \mathcal{E} \mid \|\tfrac{\mathbf{v}_b+\mathbf{v}_i}{2} - \mathbf{v}_c\| < \tfrac{\|\mathbf{v}_b-\mathbf{v}_i\|}{2}\}$$

$$\mathcal{F} := \mathcal{F} \setminus \mathcal{F}_e((\mathbf{v}_b, \mathbf{v}_i)) \quad , \ \forall (\mathbf{v}_b, \mathbf{v}_i) \in \mathcal{E}_o$$

$$\mathcal{E} := \mathcal{E} \setminus \mathcal{E}_o$$

GSRM post-processing

The topology updates of GSRM leave several higher-order polygons untriangulated, similar to NG, TRN, and GNG. In order to triangulate the constructed mesh, do Rêgo et al. [41] execute a post-processing step after the learning loop has finished. This post-processing step is similar to the one that was used to created the edges in NG. Basically, the GSRM topology and surface updates are reused without the edge aging and removal steps. However, as in NG, this CHL-based topology learning still leaves some higher-order polygons untriangulated. These are triangulated by adding suitable triangle fans.

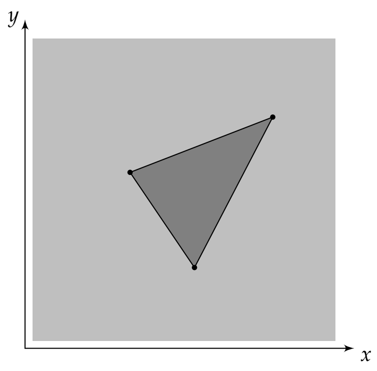
⟨*GSRM post-processing*⟩ ≡ ← p. 51
  ⟨*GSRM topology and surface updates* → p. 53⟩*
  Triangulate higher-order polygons

Examples for GSRM learning

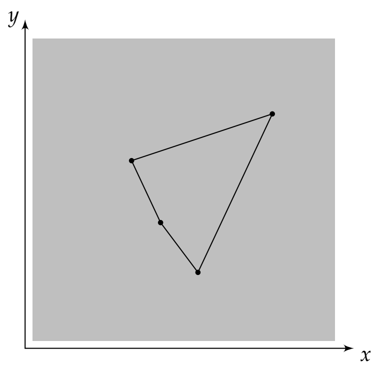
Fig. 2.20 shows an example of an initial GSRM configuration. The network initially contains three vertices that are placed at the position of three randomly selected input points. The learning parameters that are used in the reconstruction examples are set to the values that do Rêgo et al. [41] used (Tab. 2.6). The post-processing step is not used in the following examples making the results comparable to SGNG that is proposed later in this dissertation (Ch. 4), and that is based on GSRM.

In the first example the input points $\mathcal{P} \subset \mathcal{P}_\square$ (Eq. 2.1) are drawn uniformly at random from a square. Fig. 2.21 shows the resulting mesh after different numbers of iterations. During the first 99 iterations the three initial vertices get connected by edges. Once the edge loop is closed, a triangular face is added (dark gray) (Fig. 2.21(a)). One edge is split once $t \geq \lambda_s$. Due to the split the triangle is removed leaving an untriangulated four-loop of edges (Fig. 2.21(b)). Afterwards,

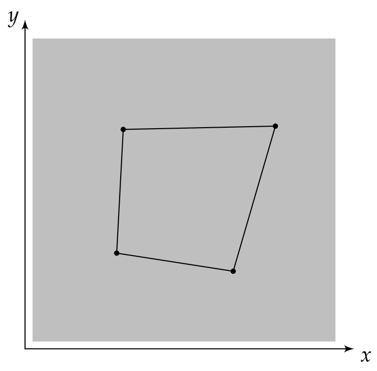
* Without edge aging and removal.



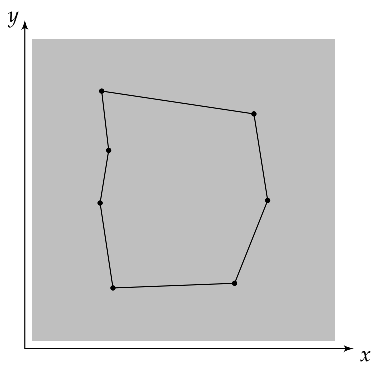
(a) $t = 99$ iterations



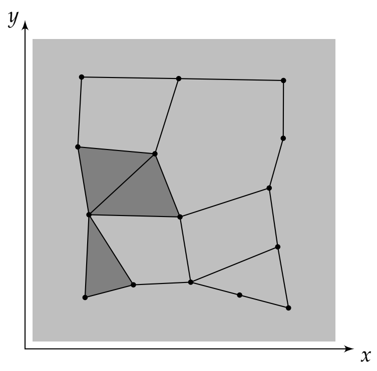
(b) $t = 110$ iterations



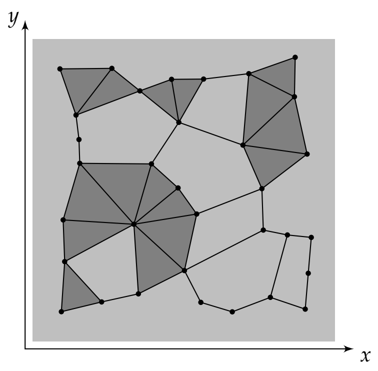
(c) $t = 199$ iterations



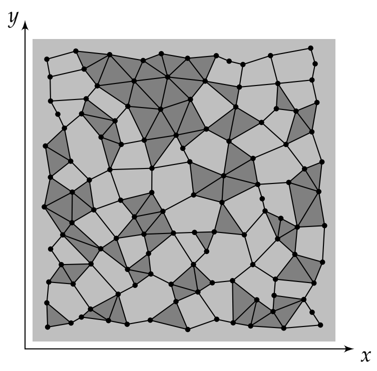
(d) $t = 480$ iterations



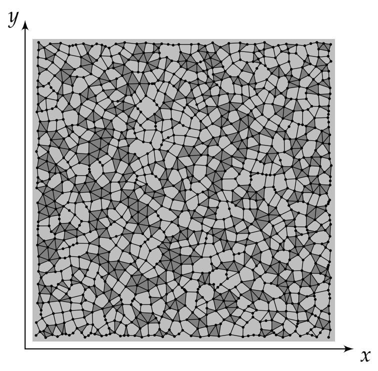
(e) $t = 1200$ iterations



(f) $t = 3000$ iterations

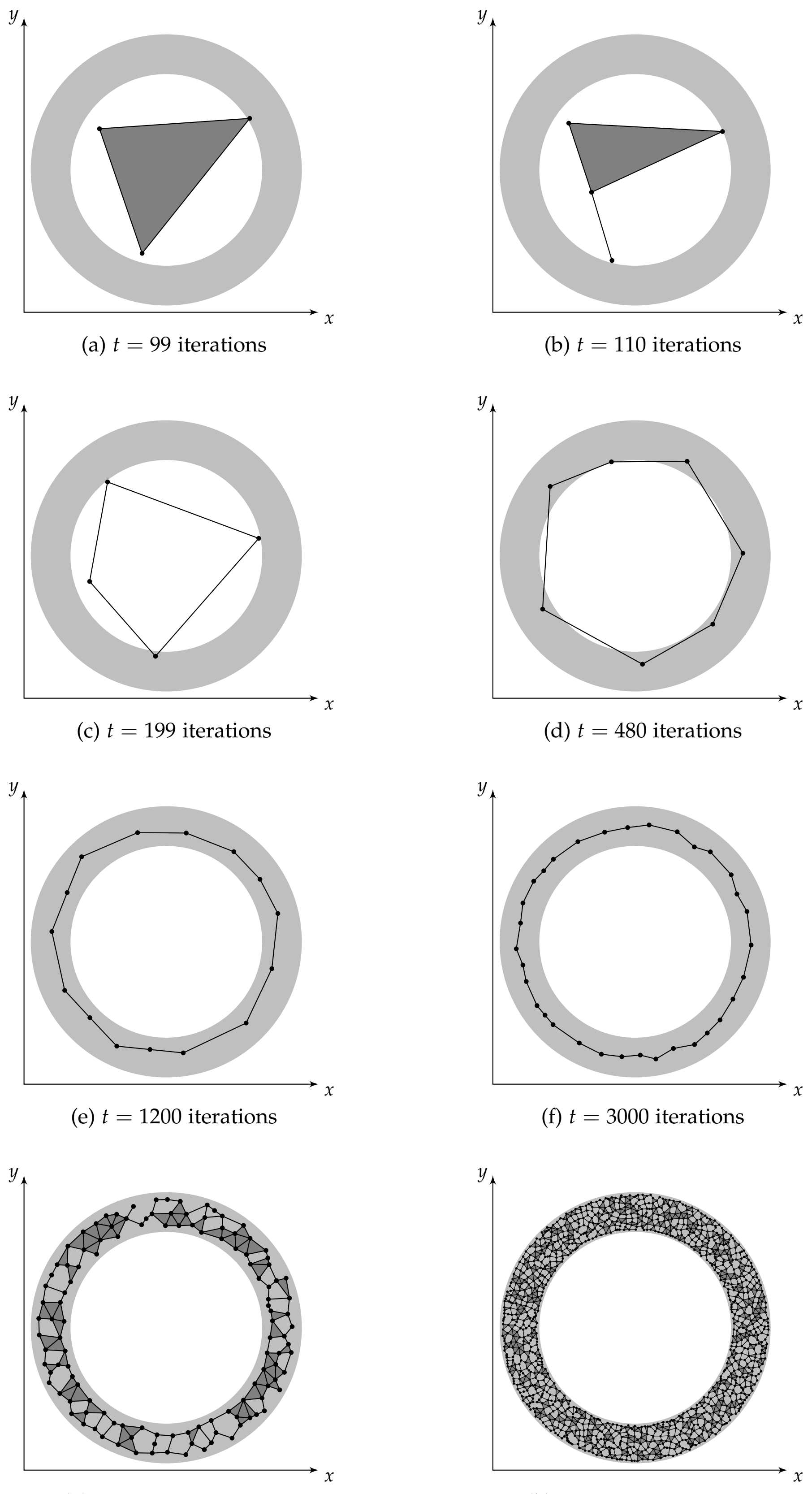


(g) $t = 12\,000$ iterations



(h) $t = 120\,000$ iterations

Figure 2.21: Gsrm reconstructing a square.



(a) $t = 99$ iterations

(b) $t = 110$ iterations

(c) $t = 199$ iterations

(d) $t = 480$ iterations

(e) $t = 1200$ iterations

(f) $t = 3000$ iterations

(g) $t = 12\,000$ iterations

(h) $t = 120\,000$ iterations

Figure 2.22: Gsrm reconstructing an annulus.

the constructed mesh is refined repeatedly, while the vertices spread out and new edges are added (Fig. 2.21(c), (d)). Eventually, edges are added that create loops of three edges and thus triangles (Fig. 2.21(e)–(g)). Finally, the vertices are distributed fairly evenly across the input square, but only few triangles are created during learning (Fig. 2.21(h)). Thus, post-processing is inevitable for GSRM. Nevertheless, most of the triangles that are created during learning are of similar size with only few irregularities.

In the second example the input points $\mathcal{P} \subset \mathcal{P}_{\circledcirc}$ (Eq. 2.2) are drawn uniformly at random from an annulus. Fig. 2.22 shows the resulting mesh after different numbers of iterations. The characteristics of the results are very similar to the square example: The triangle created during early iterations is split and deleted (Fig. 2.22(a), (b)), and a ring of connected vertices is created (Fig. 2.22(c)–(f)). Eventually, the vertices spread out and triangles and higher-order polygons that remain untriangulated are created (Fig. 2.22(g), (h)). As in the previous example, only few triangles are created during learning.

3 GROWING CELL STRUCTURES LEARN PROGRESSIVE MESHES[*]

Several related artificial neural networks that can be used to reconstruct a surface from an unstructured point cloud have been presented in the previous chapter. One of them, *growing cell structures* (GCS), automatically adapts the density of the constructed mesh to the density of the input points by applying *edge split* and *edge collapse* operations (Sec. 2.6). Ivrissimtzis et al. [70] replaced the former operation with the very closely related *vertex split* in order to improve the quality of the constructed mesh. The same pair of operations—*vertex split* and *edge collapse*—is used when generating a *progressive mesh* (PM) proposed by Hoppe [62]. Thus, GCS and PM are related.

This relationship is examined in this chapter, and the learning algorithm of GCS is modified using a binary tree of *vertex split* operations in order to create a PM directly. Thus, the constructed triangle meshes share the features of a PM, e.g., continuous level of detail, geomorphs, and compact storage, at any time during learning. Additionally, the binary tree induces a *bounding volume hierarchy* (BVH) for the nearest neighbor search that is required in GCS. Its performance is similar to the performance of an octree that is commonly used in GCS.

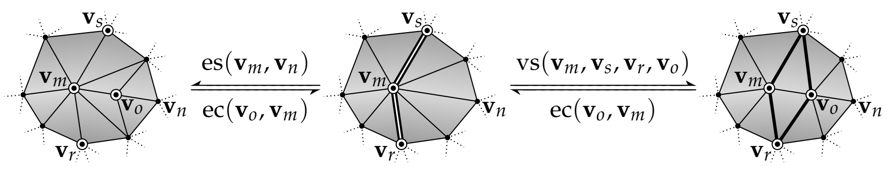
3.1 PROGRESSIVE MESHES

At first, three operations *edge split*, *vertex split*, and *edge collapse* are examined closely.

Edge split

Ivrissimtzis et al. [70] note that the *edge split* that is used in Fritzke's GCS [50, 51] does not distribute the valences of the vertices well. Therefore, they replace the *edge split* with a *vertex split*. Fig. 3.1 illustrates the difference between the two: For an *edge split* (Fig. 3.1, center to left)

[*] Parts of this chapter have been published in [121] Tom Vierjahn, Guido Lorenz, Sina Mostafawy, and Klaus Hinrichs. Growing cell structures learning a progressive mesh during surface reconstruction – a top-down approach. In *Eurographics 2012 - Short Papers*, 2012. doi: 10.2312/conf/EG2012/short/029-032.



$\mathbf{v}_s$
$\mathbf{v}_m$
$\mathbf{v}_o$
$\mathbf{v}_n$
$\mathbf{v}_r$
$\mathrm{es}(\mathbf{v}_m, \mathbf{v}_n)$
$\mathrm{ec}(\mathbf{v}_o, \mathbf{v}_m)$
$\mathrm{vs}(\mathbf{v}_m, \mathbf{v}_s, \mathbf{v}_r, \mathbf{v}_o)$
$\mathrm{ec}(\mathbf{v}_o, \mathbf{v}_m)$

Figure 3.1: *Edge split* es(·) and *edge collapse* ec(·) versus *vertex split* vs(·) and *edge collapse* operations.

in the original GCS the longest edge emanating from $\mathbf{v}_m$, i.e., $(\mathbf{v}_m, \mathbf{v}_n)$, is split by adding a new vertex $\mathbf{v}_o$ in the middle of the edge. For this purpose the edge and its adjacent triangles are deleted. Afterwards, new edges are inserted so that the new vertex $\mathbf{v}_o$ is connected to $\mathbf{v}_m$ and $\mathbf{v}_n$, and to their common neighbors. Finally, a triangle is created for each loop of three edges that contains $\mathbf{v}_o$. Thus, the valence of $\mathbf{v}_m$ does not change during an *edge split*, the valence of each of the common neighbors of $\mathbf{v}_m, \mathbf{v}_n$ is incremented by one, and the new vertex $\mathbf{v}_o$ has a valence of 4, whereas a valence of 6 is desirable in order to create a mesh containing regular triangles.

Vertex split

The *vertex split* that is used by Ivrissimtzis et al. [70] (Fig. 3.1, center to right) also adds a new vertex $\mathbf{v}_o$. This time, intuitively, by duplicating the edge strip $(\mathbf{v}_r, \mathbf{v}_m), (\mathbf{v}_m, \mathbf{v}_s)$—indicated by the double line in Fig. 3.1 (center)—and assigning the name $\mathbf{v}_o$ to the copy of $\mathbf{v}_m$. Afterwards, $\mathbf{v}_o$ is moved to its desired position and connected to $\mathbf{v}_m$ by a new edge. Finally, new triangles $\triangle(\mathbf{v}_m, \mathbf{v}_o, \mathbf{v}_s)$ and $\triangle(\mathbf{v}_m, \mathbf{v}_r, \mathbf{v}_o)$ are inserted to close the respective edge loops. By carefully selecting $\mathbf{v}_r$ and $\mathbf{v}_s$ in such a way that the star of edges emanating from $\mathbf{v}_m$ is approximately split in half, some of the edges that are emanating from $\mathbf{v}_m$ are transferred to $\mathbf{v}_o$. Thus, valences of $\mathbf{v}_m$ and $\mathbf{v}_o$ are balanced, their difference is at most one, and eventually the resulting triangles will be more regular.

While an *edge split* $\mathrm{es}(\mathbf{v}_m, \mathbf{v}_n)$ is unambiguously defined by specifying its source vertex $\mathbf{v}_m$ and its target vertex $\mathbf{v}_n$, a *vertex split* requires different vertices to be specified: Besides the source vertex, the vertex $\mathbf{v}_o$ that is created by the *vertex split* needs to be specified instead of the target vertex. Furthermore, the left vertex $\mathbf{v}_s$ and the right vertex $\mathbf{v}_r$, in the direction of the split from $\mathbf{v}_m$ to $\mathbf{v}_n$ (Fig. 3.1), need to be specified in order to unambiguously define a *vertex split* $\mathrm{vs}(\mathbf{v}_m, \mathbf{v}_s, \mathbf{v}_r, \mathbf{v}_o)$. Both operations $\mathrm{es}(\mathbf{v}_m, \mathbf{v}_n)$ and $\mathrm{vs}(\mathbf{v}_m, \mathbf{v}_s, \mathbf{v}_r, \mathbf{v}_o)$ are equivalent, iff $\mathbf{v}_o = \frac{1}{2}(\mathbf{v}_m + \mathbf{v}_n)$ and $\mathbf{v}_s, \mathbf{v}_r$ are the common neighbors of $\mathbf{v}_m, \mathbf{v}_n$.

Edge collapse

While in general *edge split* and *vertex split* are different, both operations share the same inverse: the *edge collapse* (Fig. 3.1, left to center, right to center). An *edge collapse* removes a vertex $\mathbf{v}_o$ from the mesh by first deleting the edge $(\mathbf{v}_o, \mathbf{v}_m)$ and its adjacent triangles. Additionally, the edges connecting $\mathbf{v}_o$ to the common neighbors of $\mathbf{v}_m, \mathbf{v}_o$ are deleted, but their adjacent triangles are preserved and attached to $\mathbf{v}_m$. Afterwards, the remaining edges emanating from $\mathbf{v}_o$ and their adjacent triangles are detached from $\mathbf{v}_o$ and attached to $\mathbf{v}_m$. Finally, the vertex $\mathbf{v}_o$ is deleted. An *edge collapse* $\text{ec}(\mathbf{v}_o, \mathbf{v}_m)$ is unambiguously defined by specifying its source vertex $\mathbf{v}_o$ and its target vertex $\mathbf{v}_m$.

Mesh transformations

Let $\mathcal{M} = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ denote a mesh consisting of a set $\mathcal{V}$ of vertices, a set $\mathcal{E}$ of edges, and a set $\mathcal{F}$ of triangles. Then, a *vertex split* $\text{vs}(\cdot)$ constitutes a transformation turning $\mathcal{M}$ into a transformed mesh $\mathcal{M}' = (\mathcal{V}', \mathcal{E}', \mathcal{F}')$ consisting of a transformed set $\mathcal{V}'$ of vertices, a transformed set $\mathcal{E}'$ of edges, and a transformed set $\mathcal{F}'$ of triangles:

$$\mathcal{M} \xrightarrow{\text{vs}(\cdot)} \mathcal{M}' \quad ,$$

or using function notation

$$\mathcal{M}' = \text{vs}(\cdot)(\mathcal{M}) \quad .$$

The same applies to an *edge collapse* $\text{ec}(\cdot)$:

$$\mathcal{M}' \xrightarrow{\text{ec}(\cdot)} \mathcal{M} \quad ,$$

or using function notation

$$\mathcal{M} = \text{ec}(\cdot)(\mathcal{M}') \quad .$$

Consequently, let $\text{ec}(\cdot) = \text{vs}^{-1}(\cdot) \iff \text{ec}^{-1}(\cdot) = \text{vs}(\cdot)$, then

$$\mathcal{M} \xrightarrow{\text{vs}(\cdot)} \mathcal{M}' \xrightarrow{\text{ec}(\cdot)} \mathcal{M} \quad , \quad \text{or} \quad \mathcal{M} = \big(\text{ec}(\cdot) \circ \text{vs}(\cdot)\big)(\mathcal{M}) \quad .$$

The application of a transformation is called *move* from $\mathcal{M}$ to $\mathcal{M}'$. Composition is not commutative. If

$$\mathcal{M} = \big(\text{ec}(\mathbf{v}_o, \mathbf{v}_m) \circ \text{vs}(\mathbf{v}_m, \cdot, \cdot, \mathbf{v}_o)\big)(\mathcal{M})$$

is defined, then

$$\big(\text{vs}(\mathbf{v}_m, \cdot, \cdot, \mathbf{v}_o) \circ \text{ec}(\mathbf{v}_o, \mathbf{v}_m)\big)(\mathcal{M})$$

is not, since $\mathbf{v}_o \notin \mathcal{M}$.

Legal moves

Hoppe et al. [65] defined a *legal move* in the context of mesh optimization to be a move that does not change the topological type of the mesh that it is applied to, i.e., that does not create holes or non-manifold edges. This definition is also used here. A *vertex split* does not change the topological type of $\mathcal{M}$ and is thus always a legal move. In contrast, an *edge collapse* may change the topological type of $\mathcal{M}$. Therefore, tests have to be performed in order to make sure that the desired *edge collapse* is a legal move.

Hoppe et al. [65] list three conditions for testing whether an *edge collapse* is a legal move. They prove the conjunction of them to be a necessary and sufficient condition for a legal move [66]. For this

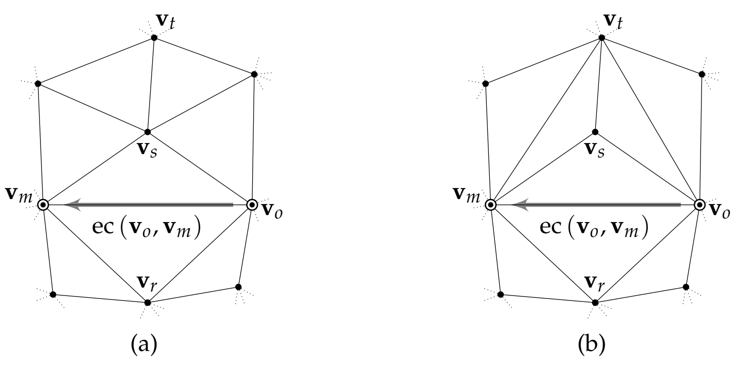


Figure 3.2: A legal (a) and an illegal (b) *edge collapse* ec $(\mathbf{v}_o, \mathbf{v}_m)$.

purpose additional definitions are made: An edge $(\mathbf{v}_m, \mathbf{v}_o) \in \mathcal{E}$ is defined to be a *boundary edge* if there exists only one $\triangle(\mathbf{v}_m, \mathbf{v}_o, \mathbf{v}_k) \in \mathcal{F}$ adjacent to it. Furthermore, a vertex $\mathbf{v}_m$ is defined to be a *boundary vertex* if there exists a boundary edge $(\mathbf{v}_m, \mathbf{v}_l) \in \mathcal{E}$. Let $\mathcal{M}$ denote a mesh with $\mathcal{M} = (\mathcal{V}, \mathcal{E}, \mathcal{F})$. Let furthermore $\mathbf{v}_m, \mathbf{v}_o \in \mathcal{V}$ and $(\mathbf{v}_m, \mathbf{v}_o) \in \mathcal{E}$. Then, an *edge collapse* $\text{ec}(\mathbf{v}_o, \mathbf{v}_m)$ that is applied to $\mathcal{M}$ in such a way that $\mathcal{M}' = \text{ec}(\mathbf{v}_o, \mathbf{v}_m)(\mathcal{M})$ with $\mathcal{M}' = (\mathcal{V}', \mathcal{E}', \mathcal{F}')$ is a legal move, iff

- $\forall \mathbf{v}_k \in \mathcal{N}_{\mathbf{v}_m} \cap \mathcal{N}_{\mathbf{v}_o} : \triangle(\mathbf{v}_m, \mathbf{v}_o, \mathbf{v}_k) \in \mathcal{F}$, and
- if $\mathbf{v}_m, \mathbf{v}_o$ are boundary vertices, $(\mathbf{v}_m, \mathbf{v}_o)$ is a boundary edge, and
- $|\mathcal{V}| > 4$ if neither $\mathbf{v}_m$ nor $\mathbf{v}_o$ are boundary vertices, or
  $|\mathcal{V}| > 3$ if either $\mathbf{v}_m$ or $\mathbf{v}_o$ are boundary vertices.

The last condition is strengthened to $|\mathcal{V}| > 4$ for practical purposes in GCS. It holds true in all but the very first iterations, since the constructed mesh grows. The second condition holds true by construction in GCS if watertight objects are reconstructed. The first condition is explicitly checked.

Fig. 3.2 shows an example for a legal and an illegal *edge collapse* $\text{ec}(\mathbf{v}_o, \mathbf{v}_m)$. The same set of vertices is used for both examples, but their connectivity is slightly different. The figure shows a part of a completely triangulated mesh, but triangles are not depicted for the sake of clarity. Thus, neither $\mathbf{v}_m$ nor $\mathbf{v}_o$ are boundary vertices, indicated by the dotted lines that are denoting further edges. The edge $(\mathbf{v}_m, \mathbf{v}_o)$ is not a boundary edge. Furthermore, $|\mathcal{V}| > 4$. Thus the second and the third conditions are satisfied.

In Fig. 3.2(a) $\mathbf{v}_m, \mathbf{v}_o$ share two common neighbors $\mathbf{v}_r, \mathbf{v}_s$. Both triangles $\triangle(\mathbf{v}_m, \mathbf{v}_o, \mathbf{v}_r), \triangle(\mathbf{v}_m, \mathbf{v}_o, \mathbf{v}_s) \in \mathcal{F}$. Thus, all three conditions hold true, and $\text{ec}(\mathbf{v}_o, \mathbf{v}_m)$ is a legal move. In contrast, in Fig. 3.2(b) $\mathbf{v}_m, \mathbf{v}_o$ share three common neighbors $\mathbf{v}_r, \mathbf{v}_s, \mathbf{v}_t$. Only the triangles $\triangle(\mathbf{v}_m, \mathbf{v}_o, \mathbf{v}_r), \triangle(\mathbf{v}_m, \mathbf{v}_o, \mathbf{v}_s) \in \mathcal{F}$, but $\triangle(\mathbf{v}_m, \mathbf{v}_o, \mathbf{v}_t) \notin \mathcal{F}$. Thus, the first condition is violated, and this time $\text{ec}(\mathbf{v}_o, \mathbf{v}_m)$ is not a legal move.

If the *edge collapse* $\text{ec}(\mathbf{v}_o, \mathbf{v}_m)$ is applied to the latter mesh anyway, duplicate coincident triangles and edges will be introduced that will most likely cause an implementation of GCS to crash. If such duplicate faces and edges are resolved correctly, the *edge collapse* will change the topological type of the mesh: Edge $(\mathbf{v}_s, \mathbf{v}_t) \in \mathcal{E}'$ will be a boundary edge, although $(\mathbf{v}_s, \mathbf{v}_t) \in \mathcal{E}$ is not. Furthermore, $(\mathbf{v}_m, \mathbf{v}_t) \in \mathcal{E}'$ will have three adjacent faces although $(\mathbf{v}_m, \mathbf{v}_t), (\mathbf{v}_o, \mathbf{v}_t) \in \mathcal{E}$ had two adjacent faces each.

Thus, whenever an inactive vertex is to be removed in GCS by an *edge collapse*, at first the set of legal moves is determined that remove the vertex. If no legal *edge collapse* exists, removal of the inactive vertex is deferred to a later iteration. Otherwise the *edge collapse* that causes the lowest regularity error is applied to the constructed mesh. See Sec. 2.6 for details.

Progressive mesh

With the above definitions available, Hoppe [62] notes that applying a sequence of *n edge collapse* operations to a triangle mesh $\mathcal{M}^n$ transforms $\mathcal{M}^n$ into a coarser mesh M^0

$$\mathcal{M}^n \xrightarrow{\mathrm{ec}_1} \mathcal{M}^{n-1} \xrightarrow{\mathrm{ec}_2} \ldots \xrightarrow{\mathrm{ec}_{n-1}} \mathcal{M}^1 \xrightarrow{\mathrm{ec}_n} \mathcal{M}^0 \quad ,$$

or using function notation

$$\mathcal{M}^0 = \big(\mathrm{ec}_n \circ \ldots \circ \mathrm{ec}_1\big)\big(\mathcal{M}^n\big) \quad ,$$

with each $\mathrm{ec}_i = \mathrm{ec}(\cdot)$ using individual vertices as source and target. Consequently, since each ec_i is invertible, any mesh $\mathcal{M}^n$ is represented by an initial, simple mesh $\mathcal{M}^0$ and a sequence of *vertex split* operations

$$\mathcal{M}^0 \xrightarrow{\mathrm{vs}_1} \mathcal{M}^1 \xrightarrow{\mathrm{vs}_2} \ldots \xrightarrow{\mathrm{vs}_{n-1}} \mathcal{M}^{n-1} \xrightarrow{\mathrm{vs}_n} \mathcal{M}^n \quad ,$$

or using function notation

$$\mathcal{M}^n = \big(\mathrm{vs}_n \circ \ldots \circ \mathrm{vs}_1\big)\big(\mathcal{M}^0\big) \quad ,$$

with each $\mathrm{vs}_i = \mathrm{vs}(\cdot)$ using individual vertices as source, new, left, and right vertex. The tuple $(\mathcal{M}^0, f)$ with $f = \mathrm{vs}_n \circ \ldots \circ \mathrm{vs}_1$ is called *progressive mesh* (PM) representation of $\mathcal{M}^n$.

A PM representation has several practical features. Geomorphs can be implemented in a straightforward way, creating smooth visual transitions from any $\mathcal{M}^i$ to any $\mathcal{M}^{i+j}$. Furthermore, meshes can be transmitted progressively and in a compressed format, e.g., from/to disk, over a network connection, or onto the GPU*. Finally, a PM allows for selective refinement and continuous level of detail.

Hoppe [62] proposes a preprocess to construct a PM based on his mesh optimization approach [65]. In that approach the sequence of *edge collapse* operations is determined by minimizing an energy function that penalizes large distance errors and large numbers of vertices, and that adds a regularization term. However, many other

* Progressive meshes are for instance included in the Microsoft® Direct3D® API [96].

error metrics that are used to find suitable simplification steps exist [57, 63, 86, 131]. Xia and Varshney introduce a hierarchical merge tree to create a PM representation of a detailed triangle mesh [132]. A precomputed PM can be refined in parallel [69], and the level of detail can be used during rendering [103]. Since GCS produces a sequence of *vertex split* operations, it is well suited to create a PM. A modification to the original learning algorithm is presented in this chapter that enables GCS to construct a PM while learning.

3.2 PROBLEM STATEMENT

Using GCS to construct $\mathcal{M}^n$ representing an original surface is regarded as a transformation

$$\mathcal{M}^n = f(\mathcal{M}^0)$$

of the initial mesh $\mathcal{M}^0$, i.e., the tetrahedron or a single triangle. However, f is then composed not only of *vertex split* operations, but also of *edge collapse* operations. Thus in general, the tuple $(\mathcal{M}^0, f)$ that is created by GCS is no PM representation of $\mathcal{M}^n$, but very similar to such a representation. *Level of detail* (LOD) can still be achieved by subsequently applying the inverted *vertex split* and *edge collapse* operations of f^{-1} to $\mathcal{M}^n = (\mathcal{V}^n, \mathcal{E}^n, \mathcal{F}^n)$. But whenever an *edge collapse* of f is inverted during the LOD steps from the fine mesh $\mathcal{M}^n$ towards the coarser mesh $\mathcal{M}^0$, the number of vertices is increased. Since these superfluous vertices do not belong to $\mathcal{V}^n$, they would have to be stored separately. To overcome this, the *edge collapse* operations of GCS are replaced by a more general vertex removal in such a way that GCS iteratively learns a PM representation.

Until the first inactive vertex $\mathbf{v}_o$ is removed from a mesh $\mathcal{M}^m$, only *vertex split* operations have been performed so far. Thus the corresponding transformation

$$f = \mathrm{vs}_m \circ \ldots \circ \mathrm{vs}_1$$

that has been recorded in order to construct $\mathcal{M}^m = f(\mathcal{M}^0)$ is a composition of *vertex split* operations. Instead of applying an *edge collapse* to $\mathcal{M}^m$ that removes $\mathbf{v}_o$, i.e., recording $\mathrm{ec}_{m+1} \circ f$, an alternative transformation f' is determined that does not produce the inactive vertex $\mathbf{v}_o$ from $\mathcal{M}^0$ in the first place. Thus, $\mathbf{v}_o$ is erased from history.

3.3 AN INTUITIVE APPROACH

Let $\mathcal{M}^m = f(\mathcal{M}^0)$ denote the triangle mesh that has been constructed so far with GCS by applying

$$f = \mathrm{vs}_m \circ \ldots \circ \mathrm{vs}_k \circ \ldots \circ \mathrm{vs}_1$$

to the initial mesh $\mathcal{M}^0$. Let furthermore $\mathbf{v}_o$ denote the inactive vertex to be removed, and let $\mathrm{vs}_k = \mathrm{vs}(\mathbf{v}_m, \mathbf{v}_s, \mathbf{v}_r, \mathbf{v}_p)$ denote the latest *vertex split* that affected $\mathbf{v}_o$ either as the source vertex, i.e., $\mathbf{v}_m = \mathbf{v}_o$, or as the new vertex, i.e., $\mathbf{v}_p = \mathbf{v}_o$. Then, f can be rewritten as

$$f = h \circ \mathrm{vs}_k \circ g \quad ,$$

where

$$\begin{aligned} g &= \mathrm{vs}_{k-1} \circ \ldots \circ \mathrm{vs}_1 \\ h &= \mathrm{vs}_m \circ \ldots \circ \mathrm{vs}_{k+1} \quad . \end{aligned}$$

In order to remove the inactive vertex from the mesh $\mathcal{M}^m$ the inverted transformation

$$(h \circ \mathrm{vs}_k)^{-1} = \mathrm{vs}_k^{-1} \circ h^{-1} = \mathrm{vs}_k^{-1} \circ \mathrm{vs}_{k+1}^{-1} \circ \ldots \circ \mathrm{vs}_m^{-1}$$

is applied to $\mathcal{M}^m$ simplifying it to $\mathcal{M}^{k-1}$ with the set $\mathcal{V}^{k-1}$ of vertices. If vs_k affected $\mathbf{v}_o$ as the new vertex, i.e., $\mathbf{v}_p = \mathbf{v}_o$, then $\mathbf{v}_o \notin \mathcal{V}^{k-1}$, since $\mathbf{v}_o$ has been removed by reverting vs_k . Thus, $\mathbf{v}_o$ is deleted.

If on the other hand vs_k affected $\mathbf{v}_o$ as the source vertex, i.e., $\mathbf{v}_m = \mathbf{v}_o$, then $\mathbf{v}_o \in \mathcal{V}^{k-1}$, and it is thus still connected in $\mathcal{M}^{k-1}$. Therefore, $\mathbf{v}_o$ cannot be deleted. Instead its neighbor $\mathbf{v}_p$ is deleted since $\mathbf{v}_p \notin \mathcal{V}^{k-1}$ after reverting vs_k. Afterwards, $\mathbf{v}_o$ takes the role of $\mathbf{v}_p$ in the mesh. For this purpose $\mathbf{v}_o$ is moved to the position of $\mathbf{v}_p$. Furthermore, the activity of $\mathbf{v}_p$ and if applicable any other attributes are transferred to $\mathbf{v}_o$. Thus, if vs_k affects $\mathbf{v}_o$ as the source vertex, removal of $\mathbf{v}_o$ is replaced by removal of its neighbor $\mathbf{v}_p$. As a consequence every reference to $\mathbf{v}_p$ in the transformations of h is replaced by a reference to $\mathbf{v}_o$.

Special care has also to be taken if no vs_k exists, since $\mathbf{v}_o \in \mathcal{V}^0$, and $\mathbf{v}_o$ is neither affected as the source vertex nor as the new vertex by any *vertex split* recorded so far. In this case f is rewritten as $f = h \circ \mathrm{vs}_1$, and f^{-1} is applied to $\mathcal{M}^m$, simplifying it to the initial mesh $\mathcal{M}^0$, with the initial set $\mathcal{V}^0$ of vertices, and $\mathbf{v}_o$ still connected in the mesh. As in the previous case, $\mathbf{v}_o$ takes the role of the new vertex that is created by the *vertex split* that is reverted last, this time vs_1.

Finally, all *vertex split* operations of h are reapplied to $\mathcal{M}^{k-1}$ in order to get back a detailed mesh $\mathcal{M}^{m'}$ reusing the original vertex positions and if applicable other attributes from the vertices in $\mathcal{V}^m$ with one caveat: Since a vertex has been deleted, some $\mathbf{v}_r$ or $\mathbf{v}_s$ might have become invalid in some $\mathrm{vs}_i = \mathrm{vs}(\mathbf{v}_m, \mathbf{v}_r, \mathbf{v}_s, \mathbf{v}_p)$ of h. Thus, these vs_i are repaired by determining new suitable right and left vertices $\mathbf{v}_r, \mathbf{v}_s$ in such a way that valences get distributed evenly. To emphasize this, the composition of the repaired operations is denoted by h'.

Let $\mathcal{M}^c = (\mathcal{V}^c, \mathcal{E}^c, \mathcal{F}^c)$ denote the mesh produced by removing $\mathbf{v}_o$ from $\mathcal{M}^m$ by an *edge collapse* as in the original GCS, and let $\mathcal{M}^{m'} = (\mathcal{V}^{m'}, \mathcal{E}^{m'}, \mathcal{F}^{m'})$ denote the mesh produced by removing $\mathbf{v}_o$ from $\mathcal{M}^m$

with the new technique. Then, $|\mathcal{V}^{m'}| = |\mathcal{V}^c|$, $|\mathcal{E}^{m'}| = |\mathcal{E}^c|$ and $|\mathcal{F}^{m'}| = |\mathcal{F}^c|$. Furthermore, the learned positions of the vertices in $\mathcal{M}^{m'}$ are the same as those in $\mathcal{M}^c$. So, $\mathcal{M}^{m'}$ is similar to $\mathcal{M}^c$ and the transformation $h' \circ g$ is the desired one. By construction all operations of $h' \circ g$ are *vertex split* operations, and thus the tuple $\left(\mathcal{M}^0, (h' \circ g)\right)$ is a PM representation of $\mathcal{M}^{m'}$.

3.4 SPLIT TREE

GCS with the modified vertex removal from the previous section creates a composition of *vertex split* operations $\text{vs}_i = \text{vs}(\mathbf{v}_m, \mathbf{v}_r, \mathbf{v}_s, \mathbf{v}_p)$ refining a coarse, initial mesh to a detailed one. Connecting the individual operations of that composition with respect to their source vertices $\mathbf{v}_m$ and their new vertices $\mathbf{v}_p$ creates a binary tree. This tree is automatically learned during the modified GCS surface reconstruction process and is referred to as a *split tree*. Since it contains only *vertex split* operations by definition, it is—in combination with an initial mesh—a representation of a detailed mesh that is equivalent to a PM.

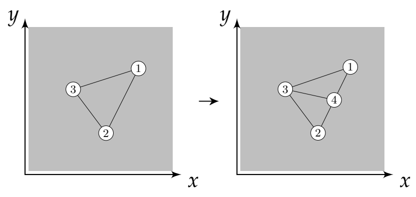
Basically, the split tree is related to the merge tree that Xia and Varshney [132] use. However, in contrast to their approach, the split tree is constructed top-down in the modified GCS. Furthermore, it uses the *vertex split* operations as internal nodes connected by edges that represent the vertices. Therefore, a split tree constitutes an operator tree. The vertices of the initial mesh $\mathcal{M}^0$ are represented by the edges leaving the root node R of the tree, whereas the vertices of the detailed mesh $\mathcal{M}^n$ are represented by the leaf nodes of the tree. Thus, the root node is i-ary, with $i = |\mathcal{V}^0|$ the number of initial vertices in $\mathcal{M}^0$. Fig. 3.3 shows the first 8 *vertex split* operations that are executed in the GCS example (Fig. 2.15). Fig. 3.4 shows the corresponding split tree.

3.5 THE OPTIMIZED APPROACH

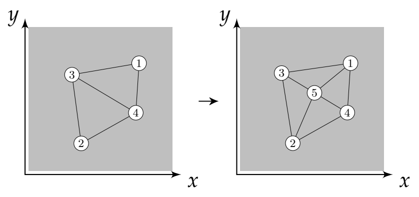
In Sec. 3.3 an intuitive approach to letting GCS learn a PM has been presented. A closer examination of the algorithm indicates that there is room for optimization leading to a more efficient algorithm: In the intuitive approach an inactive vertex $\mathbf{v}_o$ is removed by reverting all recent *vertex split* operations up to the latest vs_k that affected $\mathbf{v}_o$ as the source or the new vertex. Afterwards, all operations but vs_k are reapplied. That way, many vertices, edges, and triangles are disconnected and reconnected, and running time depends on the number of operations that have been recorded after vs_k. To optimize vertex removal, some key observations are used. These observations and the drawn conclusions that are leading to the desired optimizations are presented in the following.

Local influence

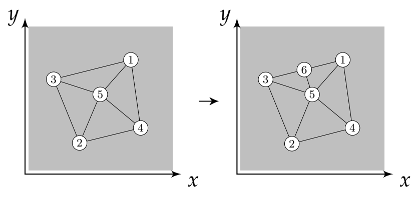
Both *vertex split* and *edge collapse* have only local influence on the mesh. During a *vertex split* a new vertex is inserted in the middle



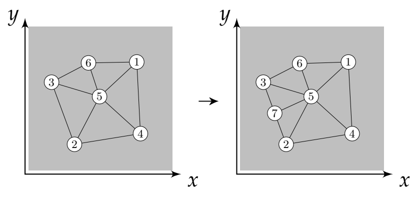
(a) $\mathrm{vs}_1 = \mathrm{vs}(\mathbf{v}_2, \mathbf{v}_3, \varepsilon, \mathbf{v}_4)$



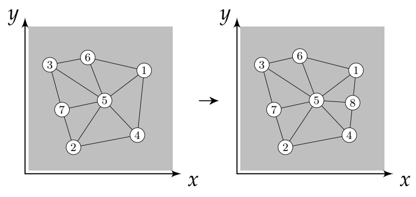
(b) $\mathrm{vs}_2 = \mathrm{vs}(\mathbf{v}_4, \mathbf{v}_2, \mathbf{v}_1, \mathbf{v}_5)$



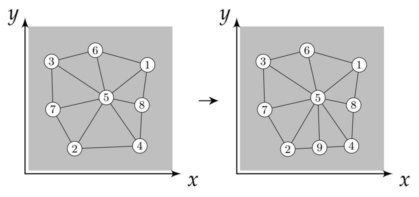
(c) $\mathrm{vs}_3 = \mathrm{vs}(\mathbf{v}_3, \varepsilon, \mathbf{v}_5, \mathbf{v}_6)$



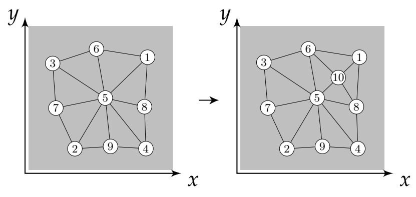
(d) $\mathrm{vs}_4 = \mathrm{vs}(\mathbf{v}_3, \mathbf{v}_5, \varepsilon, \mathbf{v}_7)$



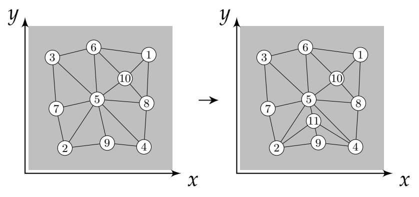
(e) $\mathrm{vs}_5 = \mathrm{vs}(\mathbf{v}_1, \varepsilon, \mathbf{v}_5, \mathbf{v}_8)$



(f) $\mathrm{vs}_6 = \mathrm{vs}(\mathbf{v}_2, \mathbf{v}_5, \varepsilon, \mathbf{v}_9)$



(g) $\mathrm{vs}_7 = \mathrm{vs}(\mathbf{v}_1, \mathbf{v}_8, \mathbf{v}_6, \mathbf{v}_{10})$



(h) $\mathrm{vs}_8 = \mathrm{vs}(\mathbf{v}_9, \mathbf{v}_2, \mathbf{v}_4, \mathbf{v}_{11})$

Figure 3.3: The first 8 *vertex split* operations that are executed in the GCS example (Fig. 2.15). The index i of each vertex $\mathbf{v}_i$ is printed in the circles that are representing the vertices. A non-existent left or right vertex is denoted by ε.

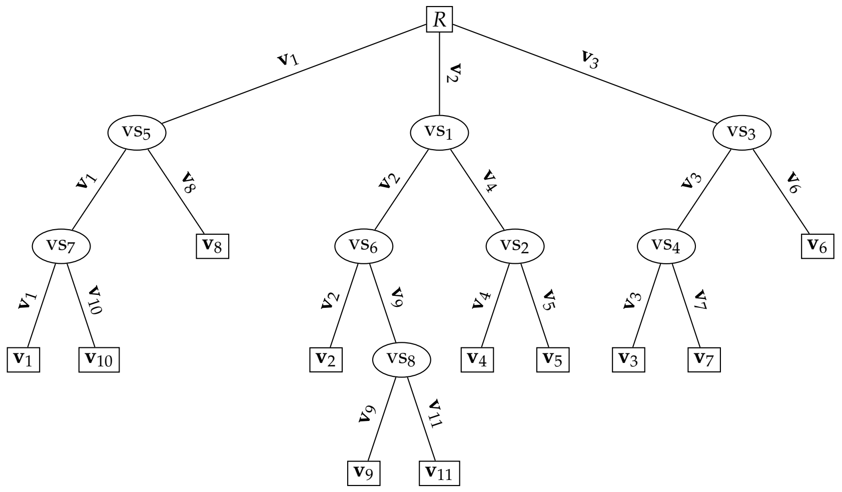


Figure 3.4: The corresponding split tree.

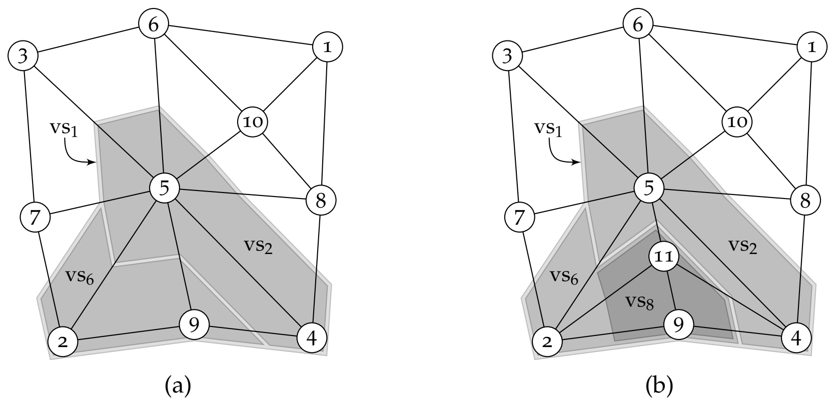


Figure 3.5: The regions of neighborhood defined by branch vs$_1$ of the split tree immediately before (a) and after (b) vs$_8$ is applied to the mesh.

of the edge connecting a source vertex $\mathbf{v}_m$ to one of its neighbors (cf. Sec. 3.1). Thus, by construction, the triangle mesh is modified only inside the polygon that is bounded by the directly connected neighbors of $\mathbf{v}_m$ (Fig. 3.1). The same applies to an *edge collapse*, since it is the inverse of a *vertex split*. Thus, removing an inactive vertex influences the mesh only locally.

Consequently, each branch of the learned split tree defines a local, continuous, hole-free region of neighborhood on the surface of the constructed mesh. Fig. 3.5 shows these regions of neighborhood for branch vs$_1$ of the previous example's split tree. A region defined by one branch can only be subdivided by *vertex split* operations that are added to the same branch. It cannot be subdivided by adding a *vertex split* to a different branch. The region that corresponds to vs$_6$ in Fig. 3.5(a), for instance, contains vertices $\mathbf{v}_2$ and $\mathbf{v}_9$. Only a *vertex split* affecting $\mathbf{v}_2$ or $\mathbf{v}_9$ as the source vertex subdivides the region, i.e., vs$_8$ in this case (Fig. 3.5(b)). *Vertex split* operations affecting a vertex as the source vertex that lies outside a region only move the boundary of the respective region. In this example, applying vs$_8$ moves the boundary of the region corresponding to vs$_2$ and causes it to shrink.

From these regions of neighborhood and the local influence of the operations a first optimization strategy is devised: Let vs$_k$ denote the latest *vertex split* that affects the vertex to be removed either as the source vertex or as the new vertex. Then, only those operations need to be reverted that are located in the branch below vs$_k$.

Immediate revert

The first optimization strategy reduces the number of operations that are reverted and reapplied, and thus reduces the number of edges and triangles that are disconnected and reconnected. However, in the worst case, vs$_k$ might be the child of the root node of the split tree. Thus still a significant number of operations is reverted and reapplied.

In total, the intuitive approach and the first optimization strategy both remove a single *vertex split* from the split tree. Since this operation has only local influence, a second optimization strategy is devised: Let vs_k again denote the latest *vertex split* that affects the vertex to be removed either as the source vertex or as the new vertex. Then, vs_k might be invertible and thus removed from the split tree directly. However, in practice, many vertices and thus the respective *vertex split* cannot be removed, since the respective reversal is illegal. Thus, the respective removal is deferred until the operation has become legal.

More candidates

Deferring removal reduces the quality of the mesh. In order to improve the quality a third optimization strategy is devised: Inactivity is caused by a triangle mesh that has become too dense so that some vertices are moved to locations where they are no longer activated by any input point. The original GCS removes the inactive vertex and thus reduces the number of vertices

It turns out that this can be addressed differently if vs_k cannot be reverted: Instead of removing the inactive vertex $\mathbf{v}_o$, it might be legal to remove one of its neighbors $\mathbf{v}_n$ by reverting the latest *vertex split* that affected $\mathbf{v}_n$ as the source vertex or as the new vertex. Afterwards, $\mathbf{v}_o$ is moved to the former position of $\mathbf{v}_n$, replacing it. That way the number of candidate reversals is increased, and it is more likely that a legal *edge collapse* is found. Thus, inactive vertices are removed earlier.

Practical consequences

These strategies lead to a working surface reconstruction algorithm, reverting only one *vertex split* during each vertex removal, directly, without disconnecting and reconnecting numerous vertices, edges and triangles. The triangle mesh is still represented by a split tree at any learning step. Unfortunately, after a certain number of vertex removals using the above algorithm, relying only on the criteria for legality by Hoppe et al. [65], the split tree cannot be reverted any more in order to get back to a coarser mesh. Thus, many desired features of the progressive mesh representation are not applicable directly anymore, e.g., continuous level of detail.

3.6 SIMULATED REVERSAL

The set of criteria whether it is legal to revert a *vertex split* defined by Hoppe et al. [65] is not applied only to a single *vertex split*, but to the complete composition of *vertex split* operations. That way the algorithm ensures that the split tree remains revertible at any time during reconstruction.

Sequence of neighbors

For any vertex $\mathbf{v}_i$ of a triangle mesh an ordered sequence $(\mathbf{n}_{\mathbf{v}_i}) = (\mathbf{v}_j, \ldots, \mathbf{v}_z)$ of neighbors $\mathbf{v}_j, \ldots, \mathbf{v}_z \in \mathcal{N}_{\mathbf{v}_i}$, i.e., vertices that are directly connected to $\mathbf{v}_i$, is determined. The elements in each sequence are ordered counter-clockwise around $\mathbf{v}_i$. For the vertices affected by the reversal of *vertex split* $\mathrm{vs}(\mathbf{v}_m, \mathbf{v}_s, \mathbf{v}_r, \mathbf{v}_o)$ (Fig. 3.1, right to center) the sequences are:

$$\begin{aligned}
(\mathbf{n}_{\mathbf{v}_m}) &= (\mathbf{v}_o, \mathbf{v}_s, \ldots, \mathbf{v}_r) \\
(\mathbf{n}_{\mathbf{v}_o}) &= (\mathbf{v}_m, \mathbf{v}_r, \ldots, \mathbf{v}_s) \\
(\mathbf{n}_{\mathbf{v}_r}) &= (\mathbf{v}_m, \ldots, \mathbf{v}_o) \\
(\mathbf{n}_{\mathbf{v}_s}) &= (\mathbf{v}_m, \mathbf{v}_o, \ldots) \\
(\mathbf{n}_{\mathbf{v}_q}) &= (\mathbf{v}_o, \ldots) \qquad , \quad \forall \mathbf{v}_q \in \mathcal{N}_{\mathbf{v}_o} \setminus \{\mathbf{v}_m, \mathbf{v}_r, \mathbf{v}_s\} \quad .
\end{aligned}$$

Let

$$\begin{aligned}
(\mathbf{n}_{\mathbf{v}_m}) &= (\mathbf{v}_o, \mathbf{v}_s, (\widehat{\mathbf{n}}_{\mathbf{v}_m}), \mathbf{v}_r) \\
(\mathbf{n}_{\mathbf{v}_o}) &= (\mathbf{v}_m, \mathbf{v}_r, (\widehat{\mathbf{n}}_{\mathbf{v}_o}), \mathbf{v}_s) \quad ,
\end{aligned}$$

with $(\widehat{\mathbf{n}}_{\mathbf{v}_m})$ and $(\widehat{\mathbf{n}}_{\mathbf{v}_o})$ explicitly denoting the inner parts of the respective sequences.

Simulated collapse

In order to simulate the consequences of reverting the *vertex split* $\mathrm{vs}(\mathbf{v}_m, \mathbf{v}_s, \mathbf{v}_r, \mathbf{v}_o)$, new sequences are determined from the above ones by removing or replacing $\mathbf{v}_o$ in the same way that the reversal affects the neighborhood in the mesh:

$$\begin{aligned}
(\mathbf{n}'_{\mathbf{v}_m}) &= ((\widehat{\mathbf{n}}_{\mathbf{v}_o}), \mathbf{v}_s, (\widehat{\mathbf{n}}_{\mathbf{v}_m}), \mathbf{v}_r) \\
(\mathbf{n}'_{\mathbf{v}_r}) &= (\mathbf{v}_m, \ldots) \\
(\mathbf{n}'_{\mathbf{v}_s}) &= (\mathbf{v}_m, \ldots) \\
(\mathbf{n}'_{\mathbf{v}_q}) &= (\mathbf{v}_m, \ldots) \qquad , \quad \forall \mathbf{v}_q \in \mathcal{N}_{\mathbf{v}_o} \setminus \{\mathbf{v}_m, \mathbf{v}_r, \mathbf{v}_s\} \quad .
\end{aligned}$$

If afterwards an element $\mathbf{v}_k$ appears more than once in any of the above sequences $(\mathbf{n}'_{\mathbf{v}_i})$, then reverting the *vertex split* will be illegal. Otherwise, reverting the *vertex split* will be legal.

Simulated removal

Suppose that a sequence of *vertex split* operations has been recorded during GCS so that

$$\begin{aligned}
\mathcal{M}^n &= (f)(\mathcal{M}^0) = (h \circ \mathrm{vs}_k \circ g)(\mathcal{M}^0) \\
&= (\mathrm{vs}_n \circ \ldots \circ \mathrm{vs}_{k+1} \circ \mathrm{vs}_k \circ \mathrm{vs}_{k-1} \ldots \circ \mathrm{vs}_1)(\mathcal{M}^0) \quad .
\end{aligned}$$

Then, checking whether vs_k can be removed from f directly while leaving the remaining f' invertible, leads to a two-step simulation using the above algorithm: First, simulate the consequences of reverting only vs_k and check whether reversal will be legal. Second, simulate the consequences of reverting h and check after each individual operation has been reverted if it will be legal. If both are legal, revert vs_k. That way, no vertices, edges, and triangles are disconnected and reconnected, but running time still depends on the number of operations that have been recorded after vs_k, and for which reversal has to be simulated.

Reducing complexity

In order to reduce the number of operations to be simulated, the findings from the previous subsection are used. Intuitively, they suggest that only a subset of operations needs to be simulated: The *vertex split* operations in the branch below the operation to be reverted. While this works well in many cases, the resulting split tree is not revertible in general. Some more operations need to be simulated.

As noted, reverting the k-th *vertex split* $\mathrm{vs}_k = \mathrm{vs}(\mathbf{v}_m, \mathbf{v}_s, \mathbf{v}_r, \mathbf{v}_o)$ affects the set $\mathcal{N}_{\mathrm{vs}}$ of vertices, with

$$\mathcal{N}_{\mathrm{vs}} = (\mathcal{N}_{\mathbf{v}_m} \cup \mathcal{N}_{\mathbf{v}_o}) \setminus \{\mathbf{v}_m, \mathbf{v}_o\} \quad .$$

For each vertex in $\mathcal{N}_{\mathrm{vs}}$ the latest *vertex split* operation of f is determined that affected the respective vertex either as the source vertex or as the new vertex. Let $\mathcal{S}_j$ denote this set:

$$\mathcal{S}_j = \{\mathrm{vs}_j = \mathrm{vs}(\mathbf{v}'_m, \mathbf{v}'_s, \mathbf{v}'_r, \mathbf{v}'_o) \mid \mathbf{v}'_m, \mathbf{v}'_o \in \mathcal{N}_{\mathrm{vs}}\} \quad .$$

Every *vertex split* $\mathrm{vs}_j \in \mathcal{S}_j$ is affected directly by reverting vs_k. Some of the $\mathrm{vs}_j \in \mathcal{S}_j$ might be located in a branch below a parent *vertex split* vs_p with $p > k$. Let $\mathcal{S}_p$ denote the set of these parents:

$$\mathcal{S}_p = \{\mathrm{vs}_p \mid \mathrm{vs}_j \in \mathcal{S}_j \text{ in branch below } \mathrm{vs}_p, k < p < j\} \quad .$$

It has been discovered, that the set $\mathcal{S}$ of *vertex split* operations, with

$$\begin{aligned}\mathcal{S} = \{&\mathrm{vs}_i \mid \mathrm{vs}_i \text{ in branch below } \mathrm{vs}_k\} \cup \\ &\mathcal{S}_j \cup \{\mathrm{vs}_i \mid \mathrm{vs}_i \text{ in branch below } \mathrm{vs}_j \in \mathcal{S}_j\} \cup \\ &\quad \mathcal{S}_p \cup \{\mathrm{vs}_i \mid \mathrm{vs}_i \text{ in branch below } \mathrm{vs}_p \in \mathcal{S}_p\} \quad ,\end{aligned}$$

contains the set of all *vertex split* operations that might not be revertible anymore after directly removing vs_k from the split tree. By construction, they are either directly affected by removing vs_k or indirectly affected by reverting h after removing vs_k.

Using $\mathcal{S}$, the number of *vertex split* operations that are simulated is reduced: The optimized algorithm from the previous subsection is applied only to those operations of h that are members of $\mathcal{S}$. Since $\mathcal{S}$ contains only the nodes of a subset of branches of a binary tree, the time needed to perform a single vertex removal is reduced. It thus does not depend on the total number of operations that have been recorded after vs_k anymore, but only on the size of a subset of those operations that is smaller on average.

3.7 IMPLICIT BOUNDING VOLUME HIERARCHY

During GCS learning the best matching, i.e., closest, vertex $\mathbf{v}_\mathrm{b}$ is determined for each randomly selected input point $\mathbf{p}_{\xi_t}$ with respect to the ℓ^2-norm. Finding $\mathbf{v}_\mathrm{b}$ requires a nearest neighbor search involving all vertices in $\mathcal{V}$. The straightforward way to find $\mathbf{v}_\mathrm{b}$ uses a linear search over $\mathcal{V}$. With an increasing number of vertices this becomes infeasible.

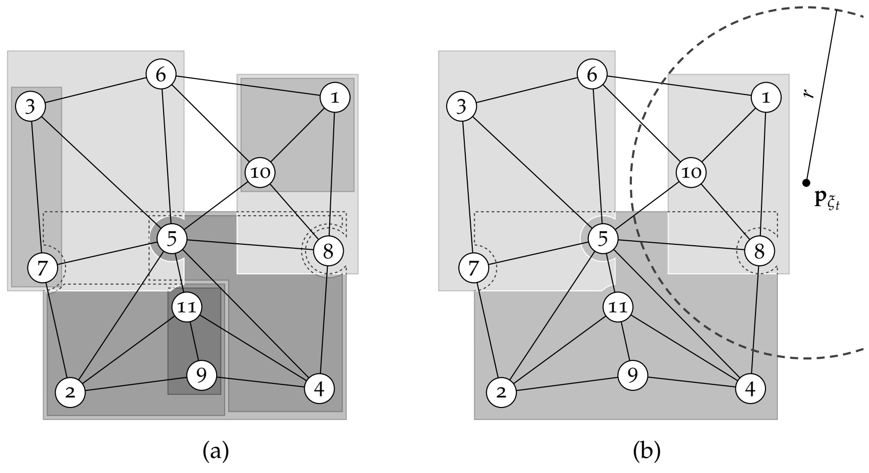


Figure 3.6: The BVH that is implicitly defined by the split tree with the AABBS slightly enlarged for the sake of clarity (a). Finding the initial radius r for the nearest neighbor search (b).

Using the split tree There are several data structures, e.g., octrees, BVHS, kd-trees [111, 135], helping efficiently search for $\mathbf{v}_b$ in a large set of vertices. All require that a separate data structure is maintained. In contrast, as noted earlier, the split tree that is created automatically during learning implicitly creates a hierarchy of regions of neighborhood. The tree stores the source vertex and the newly created vertex as children of the same operation. Since the latter vertex is always created in the immediate vicinity of the source vertex on the surface of the constructed mesh, the structure of the tree naturally suggests itself as a basis for nearest neighbor searching. Extending each internal node with a bounding box that contains all vertices in the branch below the respective node results in a suitable BVH. The split tree implicitly dictates the hierarchy of the bounding volumes so that no time consuming hierarchy creation algorithms like the one proposed by Goldsmith and Salmon [58] have to be applied. Furthermore, bottom-up or top-down approaches that are regularly used to create a BVH are not applicable in an efficient way, since new vertices are inserted frequently during GCS learning.

Creating the BVH The split tree representation of the mesh constructed so far dictates the hierarchy of bounding volumes. Thus, for hierarchy creation an AABB is added to each inner node of the tree. The AABB of each node encloses all AABB of all of its child nodes. By definition the AABB of a leaf node is set-up in such a way that it encloses only the vertex itself. The root node R of the split tree does not need any bounding box for the algorithm to work.

The hierarchy is created in an incremental way. If a *vertex split* is applied to the mesh, the AABB of the corresponding node is computed in such a way that it encloses only the source vertex and the newly created vertex. If a vertex is moved, the AABB it is contained in is updated. In both cases the AABBs of the parent nodes are updated accordingly, if necessary. Each update affects only a single path up to the root node of the split tree. Fig. 3.6a shows the BVH induced by the split tree from the previous example.

Nearest neighbor search

The best matching vertex $\mathbf{v}_\mathrm{b}$ for an input point $\mathbf{p}_{\xi_t}$ is found by a recursive branch and bound search in the BVH defined by the split tree. The algorithm traverses the split tree starting at the root node R and searches for $\mathbf{v}_\mathrm{b}$ inside a ball with radius r around $\mathbf{p}_{\xi_t}$. While searching, r is iteratively reduced. Initially, $r = \infty$. Let n denote the node that is currently processed, and let $A(n)$ denote its AABB. Let furthermore ε denote an non-existent vertex, similar to `nullptr` in C++.

```
⟨Find $\mathbf{v}_\mathrm{b}$⟩ ≡
  $r := \infty$
  $n = R$
  $\mathbf{v}_\mathrm{b} = \varepsilon$
  FINDNN($\mathbf{p}_{\xi_t}$, $r$, $n$, $\mathbf{v}_\mathrm{b}$)
```

At first, the smallest radius r of a ball around $\mathbf{p}_{\xi_t}$ is determined in such a way that the ball is just large enough to contain an AABB of one of the node's children. Let $\mathcal{B}$ yield the required ball

$$\mathcal{B} : \mathcal{P} \times \mathbb{R} \mapsto \mathbb{P}(\mathbb{R}) \quad \text{, with } \mathcal{B}(\mathbf{p}, r) = \{\mathbf{x} \in \mathbb{R}^3 \mid \|\mathbf{x} - \mathbf{p}\| \leq r\} \quad ,$$

where $\mathbb{P}(\cdot)$ denotes the power set.

```
⟨FINDNN's definition⟩ ≡
  function FINDNN(in p, inout $r$, inout $n$, inout $\mathbf{v}_\mathrm{b}$)
      for each child node $c$ of $n$ do
          $r' = \min\{d \in \mathbb{R} \mid \mathcal{B}(\mathbf{p}, d) \cap A(c) = A(c)\}$
          $r := \min(r, r')$
      end for each
```

Fig. 3.6b shows an example for the split tree's root node R and a point $\mathbf{p}_{\xi_t}$. There, the smallest ball around $\mathbf{p}_{\xi_t}$ is just large enough to contain the AABB corresponding to vs_5.

Afterwards, the AABB of each child node of vs_5 is checked if it is intersected by or contained in the ball with radius r around $\mathbf{p}_{\xi_t}$. If an intersection with the AABB of a leaf node is detected, the corresponding vertex is stored as the closest vertex found so far. Otherwise, if an intersection with a child node's AABB is detected, the respective branch is traversed as before, further reducing r and potentially finding a vertex that is located closer to $\mathbf{p}_{\xi_t}$. If no intersection has been detected, the respective branch is skipped. In the example of Fig. 3.6b the branches of vs_3 and vs_6 are skipped. Consequently, only the vertices $\mathbf{v}_1, \mathbf{v}_4, \mathbf{v}_5, \mathbf{v}_8, \mathbf{v}_{10}$ are considered during nearest neighbor search.

Table 3.1: Results of the progressive mesh experiments.

			Cathedral	Bunny	Buddha	Lucy
Median time / s	GCS	ot	258.1	199.4	241.3	192.2
	intuit.	ot	446.7	680.9	608.8	656.0
	optim.	ot	243.4	201.3	243.7	207.8
	optim.	st	250.0	208.0	252.7	237.0
Delaunay Tri. / %	GCS	ot	82.2	91.6	80.1	86.9
	intuit.	ot	76.1	70.5	49.7	50.5
	optim.	ot	79.8	91.6	78.3	82.9
	optim.	st	79.8	91.6	78.3	82.9
Val. 5–7 Vert. / %	GCS	ot	77.8	83.3	75.0	80.5
	intuit.	ot	73.7	75.3	64.1	65.6
	optim.	ot	78.8	84.0	75.1	79.2
	optim.	st	78.8	84.0	75.1	79.2

```
⟨FINDNN's definition⟩ +≡
    for each child node $c$ of $n$ do
        if $\mathcal{B}(\mathbf{p}, r) \cap A(c) \neq \emptyset$ then
            if $c$ is leaf then
                $\mathbf{v}_b$ = vertex corresponding to $c$
                $r := \|\mathbf{v}_b - \mathbf{p}\|$
            else
                FINDNN($\mathbf{p}$, $r$, $c$, $\mathbf{v}_b$)
            end if
        end if
    end for each
  end function
```

After traversing or skipping all child branches, the vertex $\mathbf{v}_b$ that is closest to $\mathbf{p}_{\xi_t}$ has been found among the vertices in the branch below the current node. Since initially the root node R and a radius of infinity are used, calling FINDNN recursively finds the best matching vertex $\mathbf{v}_b$ and its distance r to $\mathbf{p}_{\xi_t}$ efficiently by branch and bound.

3.8 RESULTS

The algorithms presented in this chapter are evaluated in terms of running times and reconstruction quality. For this purpose, the original GCS using an octree for nearest neighbor search is used as a baseline. Furthermore, the performance of the intuitive approach to PM-generating GCS using an octree for nearest neighbor search, and the performance of the optimized algorithm using either an octree or the split tree for nearest neighbor search are determined.

All techniques have been implemented in C++ and compiled with Microsoft® Visual Studio® 2010. They have been executed on a

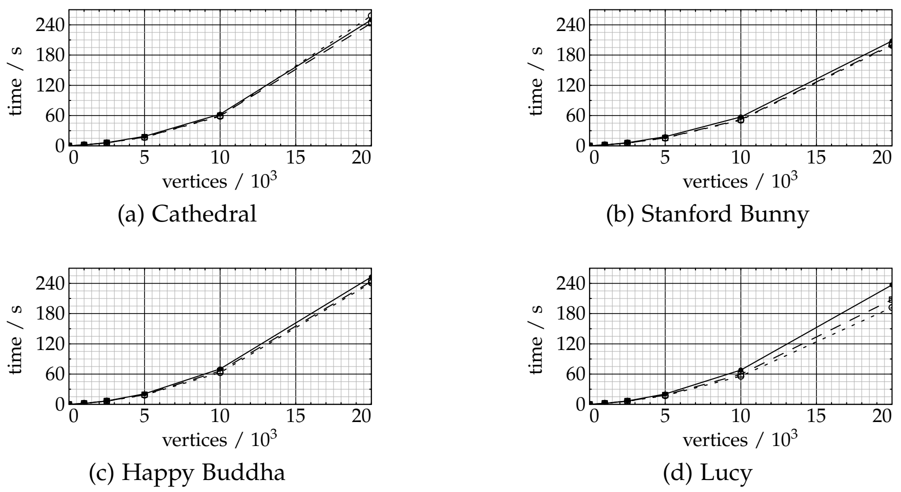


Figure 3.7: Median reconstruction times of PM-generating GCSs using the split tree (solid) and an octree (dashed) for nearest neighbor search. Original GCS as a baseline (dotted).

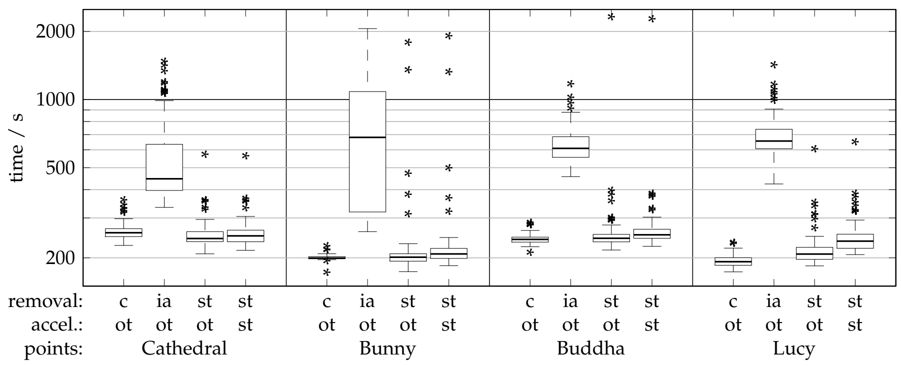


Figure 3.8: Distribution of running times. Vertex removal: original GCS (c), intuitive approach (ia), split tree (st). Acceleration structure: octree (ot) split tree (st).

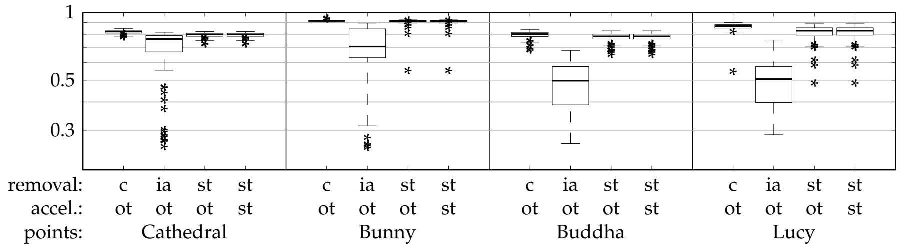


Figure 3.9: Distribution of the ratio of Delaunay triangles.

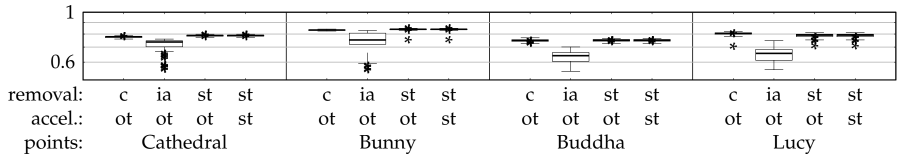


Figure 3.10: Distribution of the ratio of vertices with 5 to 7 neighbors.

Dell® workstation with a 3 GHz Intel® Xeon® x5675 CPU, 12 GB RAM running Windows® 7 Enterprise, using only a single CPU core. Meshes with 20 000 vertices have been constructed from four point clouds: Bunny, Happy Buddha, and Lucy obtained from the Stanford scanning repository [114], and a point cloud sampled from a synthetic model of the cathedral of Paderborn, Germany. Each reconstruction has been tested with 100 different random seeds.

Reconstruction times (octree)

The intuitive approach significantly reduces performance. Median reconstruction times are increased by factors between 1.7 for Cathedral and 3.4 for Bunny and Lucy (Tab. 3.1). Furthermore, the running times vary over a very wide range for Cathedral and Bunny and still over a serious range for Buddha and Lucy (Fig. 3.8).

The optimized algorithm based on a split tree effectively removes this running time overhead. Its performance is similar to the original GCS (Fig. 3.7, dotted and dashed). Median running times increased slightly by 1 % from 199.4 s using GCS to 201.3 s using the new, optimized split tree algorithm for Bunny and even decreased by 6 % from 258.1 s using GCS to 243.4 s for Cathedral. For Buddha and Lucy, median running times increased only slightly by 1 % from 241.3 s to 243.7 s for Buddha and by 8 % from 192.2 s to 207.8 s for Lucy. The box plots of Fig. 3.8 show that the interquartile ranges for GCS and the optimized split tree approach overlap. Thus, there is no significant overhead when learning a progressive mesh during surface reconstruction, directly. While deviation of running times increased significantly with the intuitive approach, deviation of running times increased only slightly for all meshes with the new optimized split tree algorithm.

Quality

Reconstruction quality is evaluated in terms of regularity. Thus, the ratio of the constructed triangles that are Delaunay triangles, and the ratio of the constructed regular vertices, i.e., vertices that have a valence in the range from 5 to 7, are determined.

Using the intuitive approach to vertex removal reduces median reconstruction quality. Both, the ratio of Delaunay triangles and the ratio of regular vertices are significantly reduced for Buddha and Lucy (Tab. 3.1). For Cathedral and Bunny both quality measurements are slightly less reduced. In contrast, the optimized approach to vertex removal using the split tree reduces median quality only slightly compared to GCS. The intuitive approach introduces a large variation in both quality measures (Fig. 3.9, 3.10). With the optimized approach quality varies in a similar range as with the original GCS, with overlapping interquartile ranges.

Split tree for nearest neighbor search

Replacing the octree by the split tree for nearest neighbor search does not impose significant running time overheads (Tab. 3.1). The interquartile ranges of the box plots overlap (Fig. 3.8). Median running times increased slightly by 3 % from 243.4 s using an Octree to 250.0 s using the BVH defined in the split tree for the cathedral, by 3 % from 201.3 s to 208.0 s for Bunny, and by 4 % from 243.7 s to 252.7 s for

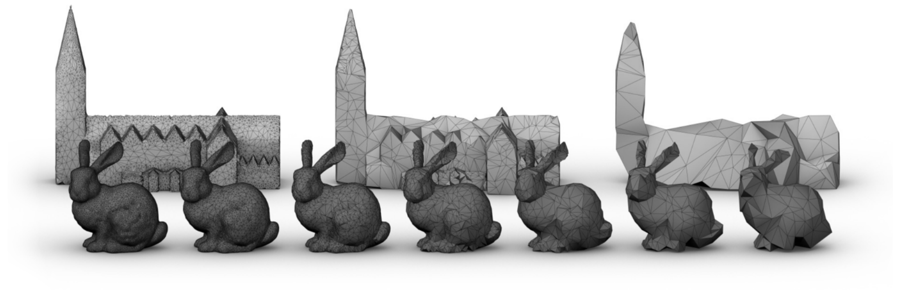


Figure 3.11: Automatically learned level of detail: Reconstruction of Stanford Bunny and Paderborn Cathedral with 20 000 triangles (far left). Using the implicit level of detail, each mesh can be simplified to, e.g., 324 triangles (far right), preserving the overall shape.

Buddha. For the high-resolution point cloud of Lucy, running times increased by 14 % from 207.8 s to 237.0 s.

The plots in Fig. 3.7 compare the running times when using an octree (dashed) and using the BVH from the split tree (solid) for all four point clouds while reconstructing different numbers of vertices. The curves are similar for all meshes. Since both Octree and BVH of the split tree find the same vertices, quality of the reconstruction does not change at all (Tab. 3.1, Fig. 3.9, 3.10).

Level of detail

With the optimized vertex removal based on the split tree, a sequence of level of detail steps is automatically learned during reconstruction. At any time, the reconstructed mesh can be reduced to a coarser mesh preserving the shape of the reconstructed object with a fairly good visual quality. However, a quadric error metric similar to that proposed by Garland and Heckbert [57] needed to be used to improve the position of the remaining vertex $\mathbf{v}_m$ after reverting a *vertex split*. Fig. 3.11 shows level of detail sequences for Stanford Bunny and Paderborn Cathedral, reducing meshes of 20 000 triangles to coarser meshes of 324 triangles.

3.9 DISCUSSION

The optimized vertex removal based on the split tree enables GCS to efficiently learn a PM during surface reconstruction. The modifications to the classical GCS algorithm do not introduce significant running time overheads. Quality in terms of regularity is only slightly reduced. However, it depends on the selected random seed, i.e., the order in which the input points are selected. Nevertheless, all the features of a PM are readily available for the reconstructed triangle mesh at any time during GCS surface reconstruction, without applying any additional processing steps afterwards.

The BVH that is defined by the split tree and is thus iteratively created during surface reconstruction allows for an efficient nearest neighbor search without having to rely on a separate data structure or on separate hierarchy creating algorithms. In practice, the binary split tree turns out to be unbalanced during GCS surface reconstruction. It is very likely that it does not define an optimal BVH as it could be achieved with other algorithms. However, experiments have demonstrated that its performance is similar to the performance of an octree that is commonly used in GCS. Thus, any further improvements to GCS that lead to a balanced or improved split tree may increase the performance of nearest neighbor search, as well.

Since the split tree is implicitly learned, neither heuristics have to be applied nor assumptions have to be made in order to determine the octree's depth and the number of vertices per cell. Thus, the BVH defined by the split tree is better suited than an octree for surface reconstruction from incrementally refined point clouds. Using the split tree for nearest neighbor search is not limited to GCS surface reconstruction. Since a split tree is an alternative representation of a PM, it is very likely useful for nearest neighbor search in any PM.

Basically, in this chapter, no new operations have been added to GCS. Instead, only the vertex removal process has been modified based on an analysis of the underlying processes. Thus, it has been demonstrated that GCS implicitly share all the features of a progressive mesh and define their own acceleration structure for nearest neighbor search. However, in the future some effort has to be spent in order to improve the quality of the PM that is learned by GCS, and in order to generalize the split tree to other types of neural networks that are used for surface reconstruction.

4 SURFACE-RECONSTRUCTING GROWING NEURAL GAS[*]

In this chapter *surface-reconstructing growing neural gas* (SGNG) is proposed, a learning-based artificial neural network that iteratively constructs a triangle mesh from a set of sample points lying on an object's surface. From these input points SGNG automatically approximates the shape and the topology of the surface. By expressing topological neighborhood via triangles, SGNG constructs a triangle mesh entirely during online learning and does not need any post-processing to close untriangulated holes. Thus, SGNG is well suited for long-running applications that require an iterative pipeline where scanning, reconstruction, and visualization are executed in parallel.

Results indicate that SGNG is a widely applicable reconstruction algorithm. It improves upon its predecessors and achieves similar or even better performance in terms of smaller reconstruction errors and better reconstruction quality than existing state-of-the-art reconstruction algorithms. If the input points are updated repeatedly during reconstruction, SGNG performs even faster than existing techniques.

4.1 ANALYSIS OF PRIOR ART

In recent approaches to surface reconstruction that are based on neural networks the fundamental algorithm that is used for approximating an original object's shape is related to Kohonen's *self-organizing map* (SOM) [81, 82, 83]. In those approaches the algorithm that is used for approximating an original object's topology is based on *competitive Hebbian learning* (CHL) [93]. A taxonomy of the fundamental neural networks with detailed descriptions of their learning algorithms is presented in Ch. 2. In this section, the CHL-based predecessors of

[*] Substantial parts of this chapter appeared in [120] Tom Vierjahn and Klaus Hinrichs. Surface-reconstructing growing neural gas: A method for online construction of textured triangle meshes. *Computers & Graphics*, pp. –, 2015. doi: 10.1016/j.cag.2015.05.016.

SGNG, are carefully analyzed in order to motivate and derive the SGNG algorithm.

Position updates The position updates, i.e., moving the closest vertex $\mathbf{v}_b$ and its directly connected neighbors $\mathbf{v}_n \in \mathcal{N}_{\mathbf{v}_b}$ towards the selected input point, let prior approaches construct a smooth approximation of the original surface's shape. These updates work well for watertight meshes and internal vertices, but hinder the predecessors of SGNG to tightly fit the boundaries of an open surface. Since the update rules

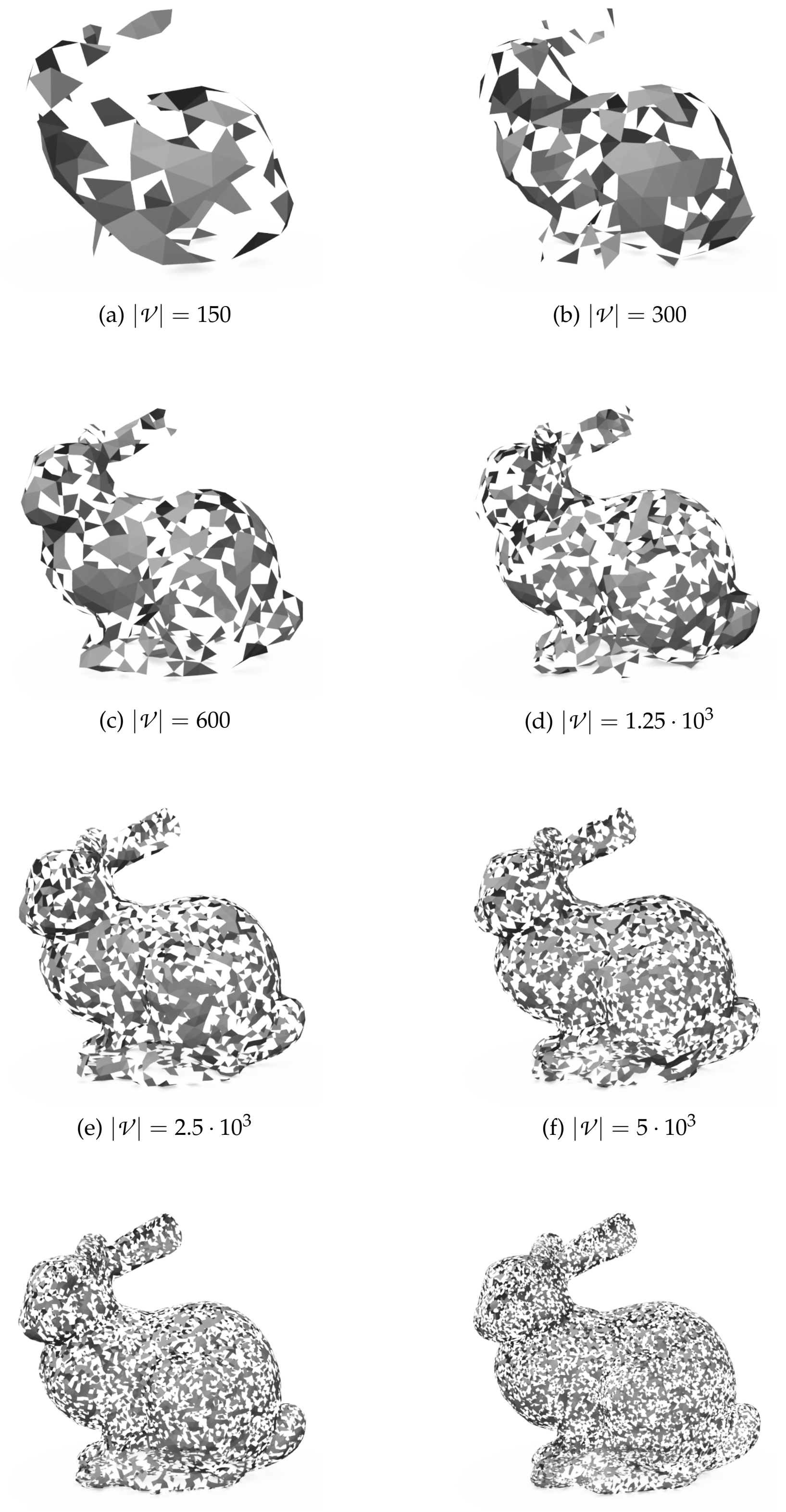
$$\begin{aligned} \mathbf{v}_b &:= \mathbf{v}_b - \beta(\mathbf{v}_b - \mathbf{p}_{\xi_t}) \\ \mathbf{v}_n &:= \mathbf{v}_n - \eta(\mathbf{v}_n - \mathbf{p}_{\xi_t}) \qquad , \ \forall \mathbf{v}_n \in \mathcal{N}_{\mathbf{v}_b} \end{aligned}$$

implement exponential moving averages, eventually each vertex is located approximately at the average position of the input points in its Voronoi cell. Consequently, many input points are lying beyond the boundaries of the constructed mesh. To overcome this, SGNG introduces an additional fitting step that moves boundary edges so that the constructed mesh represents the original surface better (Sec. 4.3).

Topology updates With its capability to create some triangles while constructing the output mesh representing an original surface, *growing self-reconstruction map* (GSRM) greatly improves over former surface reconstruction algorithms that are based on *growing neural gas* (GNG) [52] by Fritzke: Previous work for instance by Melato et al. [95] or by Holdstein and Fischer [61] did not create any triangles during learning but only during post-processing. A recent extension to GSRM [101, 102] creates even more triangles during learning.

However, using CHL and age-based edge removal in a stochastic reconstruction algorithm like the above introduces higher-order polygons to the mesh, since some edges are deleted simply by chance [53]. Thus, several holes are created during reconstruction that CHL often cannot close [31]. Their size may even increase during the subsequent iterations. In addition to that, GSRM and the recent extension [101, 102] delete obtuse triangles that contain the best and the second-best matching vertex as soon as they are detected. However, some triangles and edges may become obtuse for a few iterations in a row only due to the stochastic nature of the algorithm. Deleting them as soon as they are detected introduces additional undesired holes. Therefore, GSRM and related algorithms apply an additional topology learning phase and a post-processing step to triangulate the remaining holes that have a boundary of more than three edges. However, this is time-consuming, and it still leaves more and more holes untriangulated the sparser the input point cloud is. Finally, if continuous reconstruction and visualization is desired, it needs to be interrupted repeatedly for—possibly ineffective—post-processing.

Fig. 4.1 illustrates the iterative construction of an output triangle mesh by GSRM before post-processing is applied: From approximately



(a) $|\mathcal{V}| = 150$

(b) $|\mathcal{V}| = 300$

(c) $|\mathcal{V}| = 600$

(d) $|\mathcal{V}| = 1.25 \cdot 10^3$

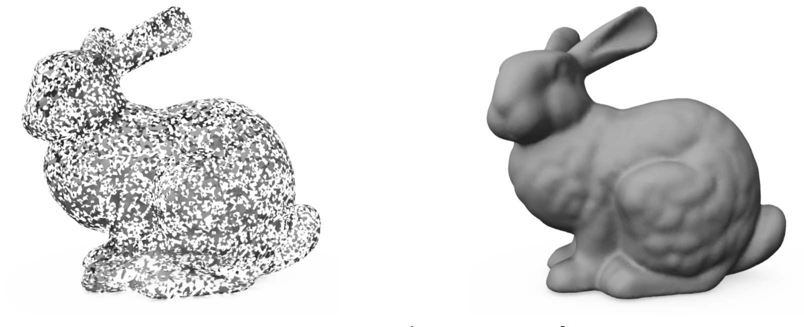
(e) $|\mathcal{V}| = 2.5 \cdot 10^3$

(f) $|\mathcal{V}| = 5 \cdot 10^3$

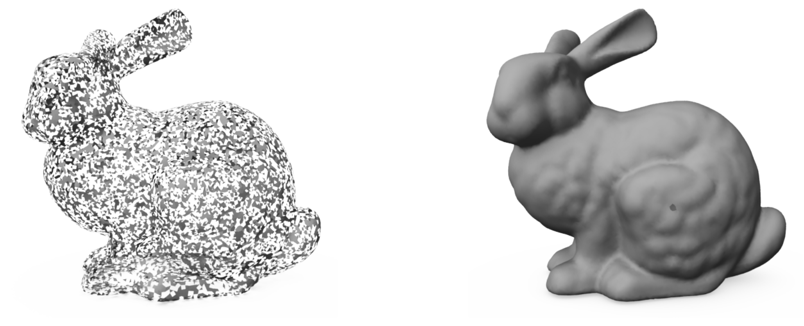
(g) $|\mathcal{V}| = 10 \cdot 10^3$

(h) $|\mathcal{V}| = 20 \cdot 10^3$

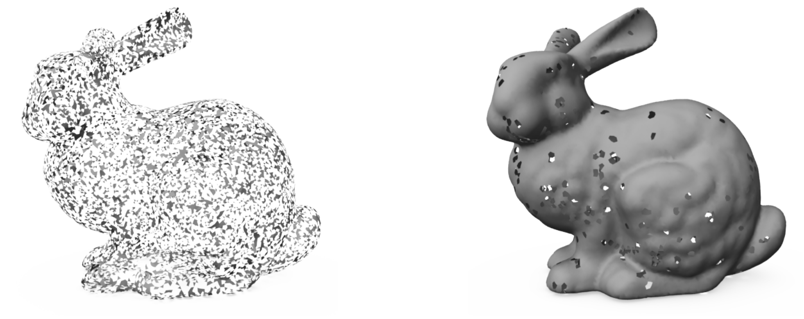
Figure 4.1: Intermediate results of GSRM reconstructions of Stanford Bunny from approximately $2.9 \cdot 10^6$ input points without post-processing.



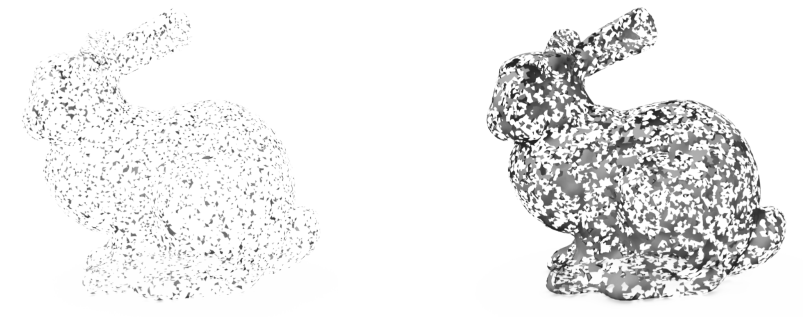
(a) $|\mathcal{P}| \approx 2.9 \cdot 10^6, |\mathcal{V}| = 20 \cdot 10^3$



(b) $|\mathcal{P}| = 1.0 \cdot 10^6, |\mathcal{V}| = 20 \cdot 10^3$



(c) $|\mathcal{P}| = 0.3 \cdot 10^6, |\mathcal{V}| = 20 \cdot 10^3$



(d) $|\mathcal{P}| = 0.1 \cdot 10^6, |\mathcal{V}| = 20 \cdot 10^3$

Figure 4.2: GSRM reconstructions of Stanford Bunny from different numbers of input points: Before (left in each subfigure) and after post-processing (right in each subfigure). The sparser the input point cloud, the more holes remain untriangulated. Results for SGNG reconstructions of same sets of input points are presented in Fig. 4.15.

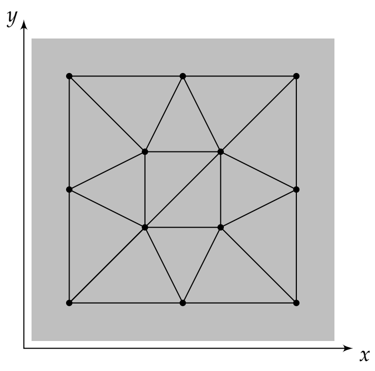
$2.9 \cdot 10^6$ input points sampled from the reconstruction of Stanford Bunny provided by the Stanford Scanning Repository [114], a rough approximation with 150 vertices is created instantly (Fig. 4.1(a)). This initial approximation gets continuously refined up to $20 \cdot 10^3$ vertices in this example (Fig. 4.1(h)). During the whole reconstruction process the constructed mesh contains many untriangulated higher-order polygons. Thus the mesh is not well suited for live visualization while reconstructing. Fig. 4.2 illustrates the effect of post-processing for GSRM reconstructions of Stanford Bunny. Post-processing is effective if the input point cloud is extraordinarily dense. Fig. 4.2(a) reproduces the results of the original experiment conducted by do Rêgo et al. [41]: There, an extremely dense point cloud with approximately $2.9 \cdot 10^6$ points [39] is used. A mesh with $20 \cdot 10^3$ vertices (left) is constructed from that point cloud. During post-processing the remaining holes are closed by triangulating the higher-order polygons (right). While this works well for the dense point cloud, post-processing gets more and more ineffective if the original object is sampled less densely (Fig. 4.2(b)–(d)): Several holes remain untriangulated even after post-processing. For reference, the original range data set of Stanford Bunny [114] contains approximately $362 \cdot 10^3$ points.

CHL in a stochastic algorithm

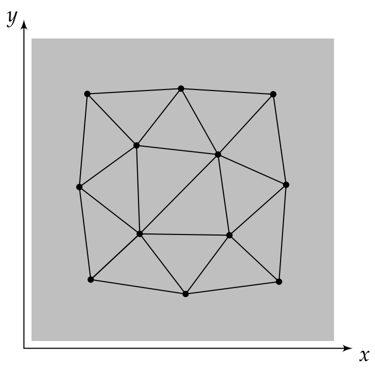
In order to derive a suitable new algorithm, the reason why GSRM leaves higher order polygons untriangulated is examined. The findings are also applicable to other techniques that use topology updates based on CHL, for instance the *topology representing network* (TRN) and GNG. For this experiment the movement and the distribution of the vertices are investigated under controlled conditions: Only the position update rules of TRN [93], GNG, and GSRM are used during the automatic iterations. Topology updates, i.e., edge removals, are performed manually. Fig. 4.3 presents the constructed mesh at different steps in order to illustrate the limitations of prior CHL-based algorithms.

Fig. 4.3(a) shows the initial mesh for this experiment. A set of 4096 input points is used that were sampled uniformly at random from a square (gray). These points are randomly presented to the neural network using epochs: Each point is presented once before any point is picked a second time. Initially and after each topology update the algorithm executes 10 epochs, i.e., 40 960 iterations, in order to reach a steady state. Since the reconstruction algorithm operates in a stochastic manner where the vertices keep moving, their mean positions during the subsequent 30 epochs, i.e., 122 880 iterations, are determined to capture trends. The figures show these mean vertex positions for clarity. The result of the first learning phase is presented in Fig. 4.3(b): The vertices have moved slightly to be approximately equally distributed across the input square.

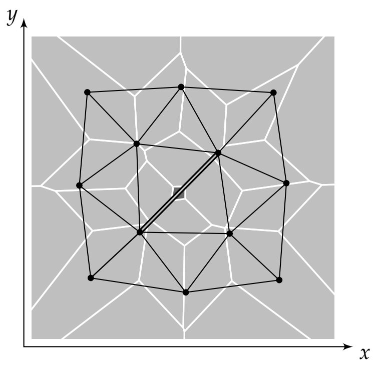
TRN, GNG, and GSRM increase the age of edges that are incident to the best matching vertex for an input point but that are not incident to the second-best matching vertex. By resetting the age of the edge



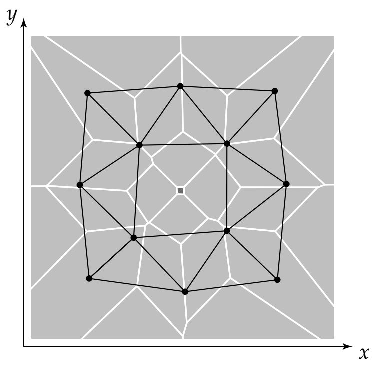
(a) Initial setup.



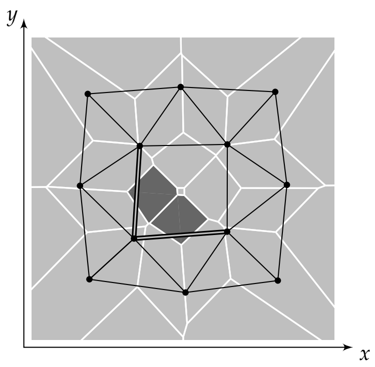
(b) First steady state.



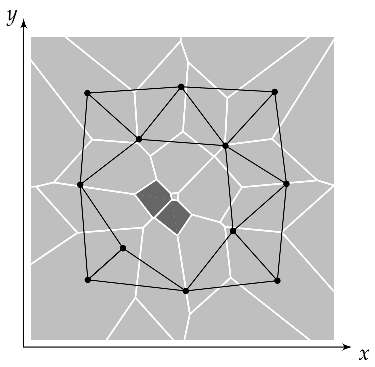
(c) First steady state with second-order Voronoi regions.



(d) Second steady state with second-order Voronoi regions.



(e) Second steady state with second-order Voronoi regions. Influence on other regions.



(f) Third steady state with second-order Voronoi regions.

Figure 4.3: Illustration of the reasons why GSRM and closely related techniques leave more and more higher order polygons untriangulated. The gray area depicts the distribution of the input points. The vertices (•) and edges (—) of the constructed mesh are printed black. The boundaries of the second-order Voronoi regions of pairs of vertices are printed white.

that is connecting the best and the second-best matching vertex, and, if necessary, by creating such an edge, the edges that are not triggered by any input point get older. They are eventually deleted once their age exceeds a predefined threshold that needs to be fine-tuned for the specific reconstruction task. See Sec. 2.5, 2.7, and 2.8 for details.

In Fig. 4.3(c) the set of input points is decomposed into the second-order Voronoi regions (enclosed by white lines) of the vertices. Points in such a region are closer to one pair of vertices than to any other vertex in the mesh. Thus, they share the same best and second-best matching vertex, or vice versa. The points from a single second-order Voronoi region therefore cause the age of a single edge to be reset. It can clearly be seen that the regions have very different sizes. Thus, assuming uniformly distributed input points, the age of some edges is likely to be reset more often than the age of other edges, depending on the size of the second-order Voronoi regions. If too low an age threshold is selected, an edge corresponding to a small second-order Voronoi region will likely be removed before it has been triggered by an input point. On the other hand, if too high a threshold is selected, the neural network will not be able to learn the topology correctly.

The second-order Voronoi regions are getting particularly small in places where vertices are locally arranged on or close to a circle. Such a configuration can be found in Fig. 4.3(c): The central region (dark gray) is by far the smallest region. Such a small region contains only very few input points, if any. Thus, the corresponding edge (thicker, double line) is very likely getting too old before it is triggered by an input point. It is thus removed creating an untriangulated quadrangle. The removal of this edge causes the now unconnected vertices to move apart during subsequent iterations (Fig. 4.3(d)) due to the update rules for the neighbors of $\mathbf{v}_b$. This may cause the corresponding second-order Voronoi region to shrink. Thus, it gets increasingly unlikely that the missing edge is recreated. In fact, in Fig. 4.3(d) the original region vanished and was replaced by the second-order Voronoi region of the other two vertices of the untriangulated quadrangle.

This effect spreads: The sizes of other regions have changed as well. In Fig. 4.3(e) the two next larger regions and their corresponding edges are highlighted. Since these regions shrank after the removal of an edge, the associated edges might now get too old due to an inadequately selected threshold or an adverse distribution of the input points. The removal of these edges turns the untriangulated quadrangle into an untriangulated hexagon, causes the lower left inner vertex to move outward in subsequent iterations, and thus causes the second-order Voronoi regions corresponding to the removed edges to shrink even further (Fig. 4.3(f)). Therefore, it becomes unlikely that the edges are recreated. This effect can be seen in the example figures that were used to illustrate TRN, GNG, and GSRM in Sec. 2.5, 2.7 and 2.8. It can also be observed in real-world reconstruction tasks. Therefore, the constructed mesh cannot be used for visualization at any time during reconstruction without applying suitable post-processing steps.

SGNG addresses this and reduces the number of untriangulated holes to a minimum during reconstruction by applying surface-aware topology updates, and by using an edge and triangle removal scheme based on more general geometric considerations that furthermore takes the stochastic nature of the algorithm into account (Sec. 4.3.2, 4.3.3). Furthermore, SGNG detects all obtuse triangles containing the best matching vertex instead of only those that contain the best and the second-best matching vertex.

Activity GSRM computes the activity $\tau(\mathbf{v}_i)$ of a vertex in terms of accumulated squared position errors. The activity of the best matching vertex $\mathbf{v}_b$ is incremented by the squared Euclidean distance of $\mathbf{v}_b$ to the currently selected input point $\mathbf{p}_{\xi_t}$:

$$\tau(\mathbf{v}_b) := \tau(\mathbf{v}_b) + \|\mathbf{v}_b - \mathbf{p}_{\xi_t}\|^2 \quad .$$

Therefore, the mesh is refined in regions where the vertices moved large distances, thus increasing the impact of noise and sparse outliers. The activity of the vertices is reduced in each iteration by a constant factor $0 < \alpha < 1$ leading to a decay over time:

$$\tau(\mathbf{v}_i) := \alpha\tau(\mathbf{v}_i) \qquad , \ \forall \mathbf{v}_i \in \mathcal{V} \quad .$$

In contrast to GSRM, SGNG adapts the density of the mesh to the density of the input points balancing the influence of noise and outliers better. For this purpose, SGNG uses a constant activity increment (Sec. 4.3.4), allowing for sorting the vertices by activity more efficiently. Furthermore, activity does not decay in SGNG, reducing complexity to be sub-quadratic. Unlike GSRM, but similar to a variant of *growing cell structures* (GCS) [51], SGNG adds vertices to the mesh by splitting the longest edge emanating from the most active vertex in order to construct more regular triangles.

Inactivity Inactivity is not addressed directly in GSRM. Instead, GSRM relies on topology updates to remove inactive vertices. If a vertex gets inactive, it is not selected as best or second-best matching vertex anymore. Thus, the age of none of the edges that are emanating from it is reset to zero anymore. Consequently, they get older and are eventually removed. Once the inactive vertex gets isolated, it is removed from the constructed mesh. Since GSRM modifies the topology in order to remove inactive vertices, additional holes are introduced that need to be closed during post-processing. Furthermore, topology learning cannot detect inactive vertices that are not connected directly to the best matching vertex, since it operates only locally for efficiency.

SGNG handles inactivity separately: Inactive vertices that are caused by too dense a reconstruction, or that are located at inversions or protuberances are removed without changing the topology of the constructed mesh (Sec. 4.3.4). For this purpose, *edge collapse* operations are used, similar to a prior extension to GCS [70]. Therefore, inactive vertices that are located in a hole of the original surface are removed while keeping the surface intact at first. The desired hole is created by topology learning in later iterations.

4.2 REQUIREMENTS AND FEATURES

The input points do not need to store any information about the surface normal for an SGNG reconstruction. The density of the input points needs to be locally homogeneous, however it may vary across the original surface. Constructed triangles will be smaller in regions with higher point density than in regions with lower density. If a triangle gets so small that no input point is located in the vicinity of its surface anymore, SGNG will remove the triangle and create a hole. In order to limit the density of the constructed mesh and thus in order to avoid erroneously untriangulated holes that are caused by too dense a reconstruction, the minimal ratio of the number of input points per constructed vertex should be limited. A minimal ratio of 4 turned out to be practical, leaving only very few holes untriangulated in regions where the vertex density exceeds the point density.

SGNG is robust to noise, if the variation of the noise is small compared to the size of the constructed triangles, otherwise overfitting will occur. Currently, estimating the amount of noise is left to the operator by setting an appropriate point to vertex ratio. Automatic noise estimation during SGNG reconstruction is left for future work.

SGNG guarantees that there will be at most two triangles adjacent to any edge. However, in order to quickly create visually meaningful, initial approximations of sparsely sampled input data, SGNG tolerates self intersections and the existence of more than one topological disk adjacent to a vertex. Since further input points can be added at any time during reconstruction, SGNG will resolve such cases automatically once the original surface is sampled densely enough. It deliberately does not guarantee that the constructed mesh will be manifold. Otherwise, for instance hourglass-like shapes needed to be sampled much more densely before reconstruction could start.

Local surface smoothness

The basic update rules for positions in GCS, GNG, and GSRM are still used in SGNG. They implement an exponential moving average of the recently presented input points. Thus, the constructed mesh eventually approximates the input points smoothly by construction.

SGNG supports this smooth approximation by constructing the surface that is smoothest whenever SGNG has to select from more than one surface configuration. By enforcing global smoothness the overall computational complexity would increase, and sharp features would be overly smoothed. However, maximizing local smoothness of two adjacent faces can be integrated in a straightforward way and is used

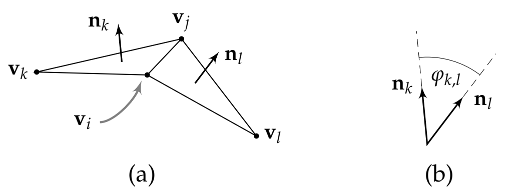


Figure 4.4: Evaluating smoothness: Unit normals $\mathbf{n}_k$, $\mathbf{n}_l$ of the adjacent triangles $\triangle(\mathbf{v}_k, \mathbf{v}_i, \mathbf{v}_j)$, $\triangle(\mathbf{v}_l, \mathbf{v}_j, \mathbf{v}_i)$ (a). Smoothness is defined in terms of the dihedral angle $\varphi_{k,l}$ (b).

as a guideline when creating triangles. Local smoothness is related to the cosine of the dihedral angle $\varphi_{k,l}$ between two triangles $\triangle(\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k)$ and $\triangle(\mathbf{v}_l, \mathbf{v}_j, \mathbf{v}_i)$ adjacent to their common edge $(\mathbf{v}_i, \mathbf{v}_j)$ (Fig. 4.4):

$$\cos \varphi_{k,l} = \mathbf{n}_k^\top \mathbf{n}_l$$

Here $\mathbf{n}_k$ and $\mathbf{n}_l$ denote the unit normals of the triangles $\triangle(\mathbf{v}_k, \mathbf{v}_i, \mathbf{v}_j)$ and $\triangle(\mathbf{v}_l, \mathbf{v}_j, \mathbf{v}_i)$, respectively

$$\begin{aligned}\mathbf{n}_k &= \frac{(\mathbf{v}_i - \mathbf{v}_k) \times (\mathbf{v}_j - \mathbf{v}_k)}{\|(\mathbf{v}_i - \mathbf{v}_k) \times (\mathbf{v}_j - \mathbf{v}_k)\|} \\ \mathbf{n}_l &= \frac{(\mathbf{v}_j - \mathbf{v}_l) \times (\mathbf{v}_i - \mathbf{v}_l)}{\|(\mathbf{v}_j - \mathbf{v}_l) \times (\mathbf{v}_i - \mathbf{v}_l)\|} \quad .\end{aligned}$$

Thus, maximizing the dihedral angle maximizes local smoothness.

A mesh for learning, a submesh for visualization

Like its predecessors SGNG creates a set $\mathcal{V}$ of vertices, a set $\mathcal{E}$ of edges, and a set $\mathcal{F}$ of faces. In contrast to its predecessors SGNG iteratively constructs a mesh that can be used for visualization at any time during reconstruction.

However, a distinction is made between the mesh $\mathcal{M}$ that is used during reconstruction and the triangle mesh $\mathcal{M}_\triangle$ to be exported, e.g., for visualization. Let $\mathcal{M} = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ contain the complete sets of faces, edges, and triangles. Then, $\mathcal{M}_\triangle = (\mathcal{V}_\triangle, \mathcal{E}_\triangle, \mathcal{F})$ contains subsets $\mathcal{V}_\triangle \subseteq \mathcal{V}$ of vertices and $\mathcal{E}_\triangle \subseteq \mathcal{E}$ that are used by the triangles in $\mathcal{F}$

$$\begin{aligned}\mathcal{V}_\triangle &= \{\mathbf{v}_i \in \mathcal{V} \mid \exists \mathbf{v}_j, \mathbf{v}_k \in \mathcal{V} : \triangle(\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k) \in \mathcal{F}\} \\ \mathcal{E}_\triangle &= \{(\mathbf{v}_i, \mathbf{v}_j) \in \mathcal{E} \mid \exists \mathbf{v}_k \in \mathcal{V} : \triangle(\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k) \in \mathcal{F}\} \quad .\end{aligned}$$

Therefore, $\mathcal{M}_\triangle$ contains only the vertices and edges that are required to create the triangles representing the original object's surface. $\mathcal{M}$ may contain (many) additional vertices and edges that are required to explore the input data, and that serve as candidates for triangle creation in later iterations in order to create a better, i.e., smoother and more regular, surface.

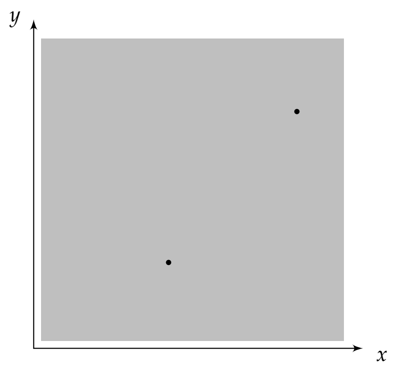


Figure 4.5: Initial configuration of SGNG reconstructing a square.

4.3 THE RECONSTRUCTION ALGORITHM

The SGNG reconstruction algorithm is based on the one used by GSRM and has been derived from the findings of the previous analysis of prior art. The new algorithm is described in this chapter. From a high-level perspective it is still similar to all the algorithms based on an artificial neural network that have been presented in Ch. 2.

$\langle$*SGNG reconstruction*$\rangle \equiv$
  $\langle$*SGNG initialization* $\twoheadrightarrow$ p. 89$\rangle$
  $\langle$*SGNG learning* $\twoheadrightarrow$ p. 90$\rangle$

SGNG initialization

SGNG reuses the initialization routine of GNG. For convenience it is repeated here: A set $\mathcal{V}$ of two vertices $\mathbf{v}_i \in \mathcal{V} \subset \mathbb{R}^3$ is created. The positions $\mathbf{v}_i$ of the vertices are initialized to the positions of two randomly selected input points (Fig. 4.5). The vertex positions will be optimized during learning. No edges are created. In addition to GNG initialization, an empty set of triangles is created. Edges and triangles will be created during learning.

$\langle$*SGNG initialization*$\rangle \equiv$ $\twoheadleftarrow$ p. 89
  $\mathcal{V} := \{\mathbf{v}_1, \mathbf{v}_2\}$, $\mathbf{v}_i := \mathbf{p}_{\xi_i} \in \mathcal{P}$, $i \in \{1,2\}$
  $\mathcal{E} := \emptyset$
  $\mathcal{F} := \emptyset$

SGNG learning

As before, online learning is iterated in a loop for a prespecified number $t_{\max}$ of iterations. Alternatively, a predefined convergence criterion can be used, for instance, a prespecified number of constructed vertices or triangles, or a prespecified maximal mean distance between the input points and the triangle mesh.

In each iteration t of SGNG learning an input point $\mathbf{p}_{\xi_t}$ is selected randomly from the set $\mathcal{P} \subset \mathbb{R}^3$ of input points. For the selected input point $\mathbf{p}_{\xi_t}$ the best matching, i.e., closest, vertex $\mathbf{v}_b \in \mathcal{V}$, and the second-best matching vertex $\mathbf{v}_c \in \mathcal{V}$ with respect to the ℓ^2-norm $\|\cdot\|$ are determined. Afterwards, the vertex positions are updated according to the position update rules of GCS. Then, the new fitting

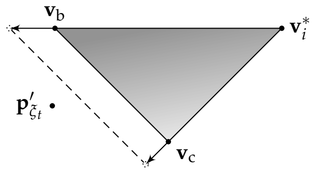


Figure 4.6: Additional fitting step introduced by SGNG.

step is used to match boundaries of the original surface more tightly and as a regularization. Finally, the topology and the density of the constructed mesh are updated according to new rules replacing the ones from GSRM.

```
⟨SGNG learning⟩ ≡                                                    ← p. 89
  for each $t \in \{1,2,\dots,\mathrm{t_{max}}\}$ do
      randomly select $\mathbf{p}_{\xi_t} \in \mathcal{P}$

      $\mathbf{v}_b = \arg\min_{\mathbf{v}_i \in \mathcal{V}} \|\mathbf{v}_i - \mathbf{p}_{\xi_t}\|$
      $\mathbf{v}_c = \arg\min_{\mathbf{v}_i \in \mathcal{V}\setminus\{\mathbf{v}_b\}} \|\mathbf{v}_i - \mathbf{p}_{\xi_t}\|$

      ⟨GCS position updates → p. 41⟩

      ⟨SGNG boundary fitting → p. 90⟩
      ⟨SGNG topology: create edges, triangles → pp. 91, 92⟩
      ⟨SGNG topology: delete edges, triangles → pp. 94, 95⟩
      ⟨SGNG density updates → p. 97⟩
  end for each
```

4.3.1 *Boundary Fitting*

If there are any triangles adjacent to the edge connecting $\mathbf{v}_b$ and $\mathbf{v}_c$, the triangle $\triangle(\mathbf{v}_b, \mathbf{v}_c, \mathbf{v}_i^*)$ whose third vertex $\mathbf{v}_i^*$ is closest to the current input point $\mathbf{p}_{\xi_t}$ is determined. Afterwards, $\mathbf{p}'_{\xi_t}$ is computed by projecting $\mathbf{p}_{\xi_t}$ into the plane through $\mathbf{v}_b, \mathbf{v}_c, \mathbf{v}_i^*$. Finally, the trilinear coordinates [128, 129] of $\mathbf{p}'_{\xi_t}$ with respect to $\triangle(\mathbf{v}_b, \mathbf{v}_c, \mathbf{v}_i^*)$ are computed. If any of them are negative, $\mathbf{p}'_{\xi_t}$ lies outside the triangle. In this case, the respective edges are moved towards $\mathbf{p}'_{\xi_t}$ by moving their vertices along the other edges of the triangle (Fig. 4.6) according to the previously used update rate β.

```
⟨SGNG boundary fitting⟩ ≡                                            ← p. 90
  if $\mathcal{F}_e((\mathbf{v}_b, \mathbf{v}_c)) \neq \emptyset$ then
      $\mathcal{V}_i = \{\mathbf{v}_i \mid \triangle(\mathbf{v}_b, \mathbf{v}_c, \mathbf{v}_i) \in \mathcal{F}_e((\mathbf{v}_b, \mathbf{v}_c))\}$
      $\mathbf{v}_i^* = \arg\min_{\mathbf{v}_i \in \mathcal{V}_i} \|\mathbf{v}_i - \mathbf{p}_{\xi_t}\|$

      $\mathbf{p}'_{\xi_t}$ = projection of $\mathbf{p}_{\xi_t}$ into plane through $\mathbf{v}_b, \mathbf{v}_c, \mathbf{v}_i^*$
      $[x \ \ y \ \ z]^\top$ = tril. coord's of $\mathbf{p}'_{\xi_t}$ w.r.t. $\triangle(\mathbf{v}_b, \mathbf{v}_c, \mathbf{v}_i^*)$,
                      s.t. $x$ corresponds to $(\mathbf{v}_b, \mathbf{v}_c)$,
                           $y$ corresponds to $(\mathbf{v}_c, \mathbf{v}_i^*)$,
                           $z$ corresponds to $(\mathbf{v}_i^*, \mathbf{v}_b)$
```

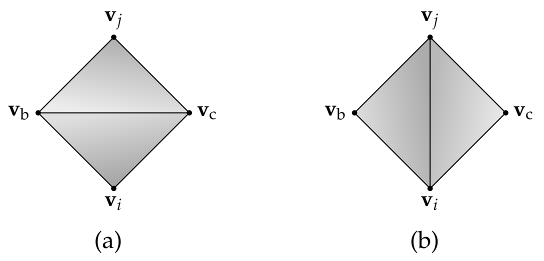


Figure 4.7: The only triangulations of a four-edge loop, requiring either edge $(\mathbf{v}_b, \mathbf{v}_c)$ (a), or edge $(\mathbf{v}_i, \mathbf{v}_j)$ (b).

```
        if $x < 0$ then
            $\mathbf{v}_b := \mathbf{v}_b - \beta \cdot x \cdot (\mathbf{v}_b - \mathbf{v}_i^*)$
            $\mathbf{v}_c := \mathbf{v}_c - \beta \cdot x \cdot (\mathbf{v}_c - \mathbf{v}_i^*)$
        end if
        ditto for $y, z$
    end if
```

4.3.2 *Topology: Create Edges, Triangles*

During learning, for a randomly selected input point the best matching, i.e., closest vertex $\mathbf{v}_b$ and the second-best matching vertex $\mathbf{v}_c$ are determined. Due to the findings from the previous analysis of prior art, SGNG connects $\mathbf{v}_b$ and $\mathbf{v}_c$ with a surface and preserves it during reconstruction. Therefore, new topology and surface updates are used. In the following pseudocode fragments $\mathcal{N}_{\mathbf{v}_b}$ denotes the set of neighbors of $\mathbf{v}_b$, and $\mathcal{N}_{\mathbf{v}_c}$ denotes the set of neighbors of $\mathbf{v}_c$. Then, the set of common neighbors of $\mathbf{v}_b$ and $\mathbf{v}_c$ is equal to the intersection $\mathcal{N}_{\mathbf{v}_b} \cap \mathcal{N}_{\mathbf{v}_c}$ of both sets. In order to later remove undesired edges and triangles, each edge $(\mathbf{v}_i, \mathbf{v}_j)$ keeps track of a penalty value

$$a_e : \mathcal{E} \mapsto \mathbb{N} \qquad , \text{ with } a_e\big((\mathbf{v}_i, \mathbf{v}_j)\big) := 0 \text{ for a new edge} \quad .$$

The basic algorithm

Assume for the moment that each edge will have at most two adjacent triangles even if new triangles are created. Later on, it is described how to enforce this by a special union operator $\Cup$.

$|\mathcal{N}_{\mathbf{v}_b} \cap \mathcal{N}_{\mathbf{v}_c}| = 0$

If $\mathbf{v}_b$ and $\mathbf{v}_c$ do not share a common neighbor, no triangles are created. Only the edge $(\mathbf{v}_b, \mathbf{v}_c)$ will be added, if it does not yet exist. Its penalty $a_e\big((\mathbf{v}_b, \mathbf{v}_c)\big)$ is reset to zero, no matter if the edge already existed or was just created.

```
⟨SGNG topology: create edges, triangles⟩ ≡                    ← p. 90
  if $\mathcal{N}_{\mathbf{v}_b} \cap \mathcal{N}_{\mathbf{v}_c} = \emptyset$ then
      $\mathcal{E} := \mathcal{E} \cup \{(\mathbf{v}_b, \mathbf{v}_c)\}$
      $a_e\big((\mathbf{v}_b, \mathbf{v}_c)\big) := 0$
```

$|\mathcal{N}_{\mathbf{v}_b} \cap \mathcal{N}_{\mathbf{v}_c}| = 1$

If $\mathbf{v}_b$ and $\mathbf{v}_c$ share one common neighbor $\mathbf{v}_i$, the edge $(\mathbf{v}_b, \mathbf{v}_c)$ and the triangle $\triangle(\mathbf{v}_b, \mathbf{v}_i, \mathbf{v}_c)$ will be created, if they do not yet exist. The penalty $a_e\big((\mathbf{v}_b, \mathbf{v}_c)\big)$ of the edge $(\mathbf{v}_b, \mathbf{v}_c)$ is reset to zero.

```
⟨SGNG topology: create edges, triangles⟩ +≡                    ← p. 90
  else if $\mathcal{N}_{\mathbf{v}_b} \cap \mathcal{N}_{\mathbf{v}_c} = \{\mathbf{v}_i\}$ then
    $\mathcal{E} := \mathcal{E} \cup \{(\mathbf{v}_b, \mathbf{v}_c)\}$
    $\mathcal{F} := \mathcal{F} \Cup \{\triangle(\mathbf{v}_b, \mathbf{v}_i, \mathbf{v}_c)\}$
    $a_e\big((\mathbf{v}_b, \mathbf{v}_c)\big) := 0$
```

$|\mathcal{N}_{\mathbf{v}_b} \cap \mathcal{N}_{\mathbf{v}_c}| \geq 2$

If $\mathbf{v}_b, \mathbf{v}_c$ share two or more common neighbors SGNG selects the two most active ones, $\mathbf{v}_i, \mathbf{v}_j$. These represent the original surface best, since they have been selected most frequently as best match to an input point. In this case, two different surface configurations exist:

- the triangles $\triangle(\mathbf{v}_b, \mathbf{v}_i, \mathbf{v}_c)$ and $\triangle(\mathbf{v}_b, \mathbf{v}_c, \mathbf{v}_j)$ (Fig. 4.7(a))
- the triangles $\triangle(\mathbf{v}_b, \mathbf{v}_i, \mathbf{v}_j)$ and $\triangle(\mathbf{v}_c, \mathbf{v}_j, \mathbf{v}_i)$ (Fig. 4.7(b))

The former requires edge $(\mathbf{v}_b, \mathbf{v}_c)$ to be present, the latter edge $(\mathbf{v}_i, \mathbf{v}_j)$. SGNG selects the edge of both candidates that creates the smoothest surface, i.e., at which the larger dihedral angle ($\varphi(\mathbf{v}_b, \mathbf{v}_c)$ or $\varphi(\mathbf{v}_i, \mathbf{v}_j)$) will be located. If necessary, the required edge and its adjacent triangles are created accordingly. $\Cup$ does not add any triangles if the corresponding required edge already has two adjacent triangles. The penalty $a_e\big((\mathbf{v}_b, \mathbf{v}_c)\big)$ or $a_e(\mathbf{v}_i, \mathbf{v}_j)$ of the edge $(\mathbf{v}_b, \mathbf{v}_c)$ or $(\mathbf{v}_i, \mathbf{v}_j)$, respectively, is reset to zero. Any existing but no longer desired edge and its adjacent triangles are deleted.

In addition to the above, SGNG detects configurations where $\mathbf{v}_b$ is part of a not yet triangulated 4-loop. If none of the edges of that 4-loop has two adjacent triangles, it will be triangulated automatically.

```
⟨SGNG topology: create edges, triangles⟩ +≡                    ← p. 90
  else
    $\{\mathbf{v}_i, \mathbf{v}_j\} = \{\mathbf{v}_i, \mathbf{v}_j \in \mathcal{N}_{\mathbf{v}_b} \cap \mathcal{N}_{\mathbf{v}_c} \mid \text{most active}\}$
    if $\varphi(\mathbf{v}_b, \mathbf{v}_c) \geq \varphi(\mathbf{v}_i, \mathbf{v}_j)$ then
      $\mathcal{E} := \mathcal{E} \cup \{(\mathbf{v}_b, \mathbf{v}_c)\}$
      $\mathcal{F} := \mathcal{F} \Cup \{\triangle(\mathbf{v}_b, \mathbf{v}_i, \mathbf{v}_c)\ ,\ \triangle(\mathbf{v}_b, \mathbf{v}_c, \mathbf{v}_j)\}$
      $a_e\big((\mathbf{v}_b, \mathbf{v}_c)\big) := 0$
    else
      $\mathcal{E} := \mathcal{E} \cup \{(\mathbf{v}_i, \mathbf{v}_j)\}$
      $\mathcal{F} := \mathcal{F} \Cup \{\triangle(\mathbf{v}_b, \mathbf{v}_i, \mathbf{v}_j)\ ,\ \triangle(\mathbf{v}_c, \mathbf{v}_j, \mathbf{v}_i)\}$
      $a_e\big((\mathbf{v}_i, \mathbf{v}_j)\big) := 0$
    end if
    delete no longer desired edge and triangles
  end if
```

The special union operator $\Cup$

The previous algorithm determines the edge that is required to connect $\mathbf{v}_b$ and $\mathbf{v}_c$ with a surface. Whenever a triangle has to be created adjacent to such a required edge in order to fill a 3-loop, one or both of

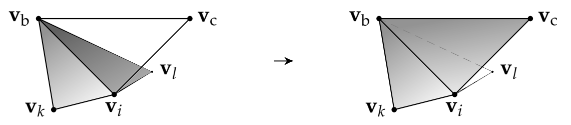


Figure 4.8: Enforcing the 2-adjacent-triangle guarantee: If triangle $\triangle(\mathbf{v}_\mathrm{b}, \mathbf{v}_i, \mathbf{v}_\mathrm{c})$ has to be created but the edge $(\mathbf{v}_\mathrm{b}, \mathbf{v}_i)$ has two adjacent triangles (left), SGNG will pick the smoothest surface configuration from all potential ones (right).

the other edges of the loop might already have two adjacent triangles. Then, creation of the new triangle would violate the guarantee that each edge has at most two adjacent triangles. Therefore, SGNG uses a union operator $\Cup$ that complies with this guarantee. $\Cup$ picks the smoothest surface configuration, i.e., the configuration with the largest dihedral angles, and it creates and removes triangles accordingly.

Assume for instance that $\mathbf{v}_\mathrm{b}$ and $\mathbf{v}_\mathrm{c}$ share one common neighbor $\mathbf{v}_i$, that the edge $(\mathbf{v}_\mathrm{b}, \mathbf{v}_\mathrm{c})$ was just created, and that the basic algorithm requests to add a new triangle $\triangle(\mathbf{v}_\mathrm{b}, \mathbf{v}_i, \mathbf{v}_\mathrm{c})$:

$$\mathcal{F} := \mathcal{F} \Cup \{\triangle(\mathbf{v}_\mathrm{b}, \mathbf{v}_i, \mathbf{v}_\mathrm{c})\} \quad . \tag{4.1}$$

Let there be no triangles adjacent to the edge $(\mathbf{v}_i, \mathbf{v}_\mathrm{c})$, but let there be two triangles $\triangle(\mathbf{v}_\mathrm{b}, \mathbf{v}_k, \mathbf{v}_i)$ and $\triangle(\mathbf{v}_\mathrm{b}, \mathbf{v}_i, \mathbf{v}_l)$ adjacent to the other edge $(\mathbf{v}_\mathrm{b}, \mathbf{v}_i)$ (Fig. 4.8 left). Creating the new triangle would add a third triangle adjacent to $(\mathbf{v}_\mathrm{b}, \mathbf{v}_i)$ and thus violate the above guarantee.

By also taking the edge $(\mathbf{v}_\mathrm{b}, \mathbf{v}_i)$ and the vertices $\mathbf{v}_k$ and $\mathbf{v}_l$ into account, three different valid surface configurations exist:

- triangles $\triangle(\mathbf{v}_\mathrm{b}, \mathbf{v}_k, \mathbf{v}_i)$ and $\triangle(\mathbf{v}_\mathrm{b}, \mathbf{v}_i, \mathbf{v}_l)$
- triangles $\triangle(\mathbf{v}_\mathrm{b}, \mathbf{v}_k, \mathbf{v}_i)$ and $\triangle(\mathbf{v}_\mathrm{b}, \mathbf{v}_i, \mathbf{v}_\mathrm{c})$
- triangles $\triangle(\mathbf{v}_\mathrm{b}, \mathbf{v}_l, \mathbf{v}_i)$ and $\triangle(\mathbf{v}_\mathrm{b}, \mathbf{v}_i, \mathbf{v}_\mathrm{c})$

SGNG evaluates the dihedral angles for all three configurations and picks the smoothest, i.e., the one which generates the largest dihedral angle—the second in case of Fig. 4.8 right. In this case the union operator $\Cup$ (Eq. 4.1) thus evaluates to

$$\mathcal{F} \Cup \{\triangle(\mathbf{v}_\mathrm{b}, \mathbf{v}_i, \mathbf{v}_\mathrm{c})\} = (\mathcal{F} \setminus \{\triangle(\mathbf{v}_\mathrm{b}, \mathbf{v}_i, \mathbf{v}_l)\}) \cup \{\triangle(\mathbf{v}_\mathrm{b}, \mathbf{v}_i, \mathbf{v}_\mathrm{c})\} \quad .$$

The number of potential surface configurations that are evaluated will increase if $\mathbf{v}_\mathrm{b}$ and $\mathbf{v}_i$ share more common neighbors. This concept is extended in a straightforward manner even to configurations where both of the existing edges $(\mathbf{v}_\mathrm{b}, \mathbf{v}_i)$ and $(\mathbf{v}_i, \mathbf{v}_\mathrm{c})$ have two adjacent triangles. Then, the sum of the respective dihedral angles is maximized.

SGNG might produce some edges that have no adjacent triangles. These edges increase the number of potential surface configurations in future iterations and thus help construct the mesh that fits the input data best. However, these edges will eventually be deleted in subsequent iterations and will not be part of the final mesh.

4.3.3 *Topology: Delete Edges, Triangles*

SGNG introduces new surface-aware removal schemes for triangles and edges. Three different, undesired geometric cases are addressed: Triangles that are covering holes in the original surface, obtuse triangles, and isolated edges, i.e., edges without adjacent triangles. Since these may be caused by the stochastic nature of the algorithm they may exist for a few iterations. SGNG penalizes undesired triangles and edges whenever they are detected instead of deleting them instantly.

Edge penalties

To enable deleting edges and thus triangles, each edge keeps track of its penalty a_e. Whenever the edge and triangle creation algorithm from the previous subsection determines that an edge is required for the selected surface configuration, its penalty is reset to zero. If an edge is determined to be undesired, its penalty is incremented by one. Once the penalty assigned to an edge exceeds a predefined threshold a_max, the edge and any triangles adjacent to it are deleted.

```
⟨SGNG topology: delete edges, triangles⟩ ≡                                   ← p. 90
  $\mathcal{E}_\mathrm{u} := \emptyset$
  ⟨SGNG detect isolated edges, update $\mathcal{E}_\mathrm{u}$ → p. 94⟩
  ⟨SGNG detect obtuse triangles, update $\mathcal{E}_\mathrm{u}$ → p. 95⟩
  $a_\mathrm{e}((\mathbf{v}_\mathrm{b},\mathbf{v}_k)) := a_\mathrm{e}((\mathbf{v}_\mathrm{b},\mathbf{v}_k)) + 1 \quad , \; \forall(\mathbf{v}_\mathrm{b},\mathbf{v}_k) \in \mathcal{E}_\mathrm{u}$
  $\mathcal{E}_{\mathrm{a}_\mathrm{max}} = \{(\mathbf{v}_\mathrm{b},\mathbf{v}_k) \in \mathcal{E} \mid a_\mathrm{e}((\mathbf{v}_\mathrm{b},\mathbf{v}_k)) > \mathrm{a}_\mathrm{max}\}$

  $\mathcal{F} := \mathcal{F} \setminus \mathcal{F}_e((\mathbf{v}_\mathrm{b},\mathbf{v}_k)) \quad , \; \forall(\mathbf{v}_\mathrm{b},\mathbf{v}_k) \in \mathcal{E}_{\mathrm{a}_\mathrm{max}}$
  $\mathcal{E} := \mathcal{E} \setminus \mathcal{E}_{\mathrm{a}_\mathrm{max}}$
  $\mathcal{V} := \mathcal{V} \setminus \{\mathbf{v}_r \in \mathcal{V} \mid \forall \mathbf{v}_s \in \mathcal{V} : (\mathbf{v}_r,\mathbf{v}_s) \notin \mathcal{E}\}$
```

Isolated edges

Isolated edges, i.e., edges with no adjacent triangles, are required especially during early iterations at the beginning of a reconstruction process or whenever new points are added to the point cloud in order to explore the input data. However, when no triangles are created adjacent to an edge, it is not required to construct a surface. Isolated edges are handled in a straightforward manner: If any edge emanating from the best matching vertex $\mathbf{v}_\mathrm{b}$ has no adjacent triangles, its penalty $a_\mathrm{e}(\cdot)$ is incremented by one.

```
⟨SGNG detect isolated edges, update $\mathcal{E}_\mathrm{u}$⟩ ≡                                   ← p. 94
  $\mathcal{E}_\mathrm{u} := \mathcal{E}_\mathrm{u} \cup \{(\mathbf{v}_\mathrm{b},\mathbf{v}_k) \in \mathcal{E} \mid$
                         $\forall \mathbf{v}_l \in \mathcal{V} : \triangle(\mathbf{v}_\mathrm{b},\mathbf{v}_k,\mathbf{v}_l) \notin \mathcal{F}\}$
```

Obtuse triangles

Obtuse triangles are handled by determining the Thales circle for any edge $(\mathbf{v}_\mathrm{b},\mathbf{v}_i)$ that is emanating from $\mathbf{v}_\mathrm{b}$ (Fig. 4.9). If another directly connected neighbor $\mathbf{v}_j \neq \mathbf{v}_i$ of $\mathbf{v}_\mathrm{b}$ lies inside that circle, the corresponding triangle $\triangle(\mathbf{v}_\mathrm{b},\mathbf{v}_i,\mathbf{v}_j)$ is obtuse, i.e., one of its angles is

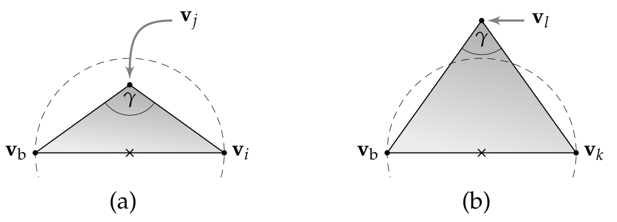


Figure 4.9: The Thales circle is used to detect obtuse triangles ($\gamma > 90°$): $\triangle(\mathbf{v}_b, \mathbf{v}_i, \mathbf{v}_j)$ is obtuse, and is thus penalized (a). $\triangle(\mathbf{v}_b, \mathbf{v}_k, \mathbf{v}_l)$ is more regular, and is thus preserved (b).

greater than 90°. The penalty $a_e\big((\mathbf{v}_b, \mathbf{v}_i)\big)$ of the edge $(\mathbf{v}_b, \mathbf{v}_i)$ that is causing the triangle to be obtuse is incremented by one. The penalty is incremented even if $\triangle(\mathbf{v}_b, \mathbf{v}_i, \mathbf{v}_j) \notin \mathcal{F}$, since the respective triangle might be created in subsequent iterations. That way SGNG later constructs more regular triangles.

```
⟨SGNG detect obtuse triangles, update $\mathcal{E}_u$⟩ ≡                ← p. 94
  for each $\mathbf{v}_i \in \mathcal{N}_{\mathbf{v}_b}$ do
    if $\exists \mathbf{v}_j \in \mathcal{N}_{\mathbf{v}_b} \setminus \{\mathbf{v}_i\} : \|\mathbf{v}_j - \frac{\mathbf{v}_b+\mathbf{v}_i}{2}\| < \|\mathbf{v}_b - \frac{\mathbf{v}_b+\mathbf{v}_i}{2}\|$ then
      $\mathcal{E}_u := \mathcal{E}_u \cup \{(\mathbf{v}_b, \mathbf{v}_i)\}$
    end if
  end for each
```

Triangle penalties

Triangles that are covering holes in the original surface cannot be removed by removing one of its edges, since the edge might be necessary to represent the boundary of the original surface. Therefore, each triangle $\triangle(\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k)$ keeps track of a separate penalty value

$$a_\triangle : \mathcal{F} \mapsto \mathbb{N} \quad , \text{ with } a_\triangle\big(\triangle(\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k)\big) := 0 \text{ for a new triangle} \quad .$$

In contrast to the penalty assigned to the edges the penalty assigned to the triangles may not be reset to zero. Otherwise triangles that are covering holes will not be detected robustly. Once the penalty assigned to a triangle exceeds a predefined threshold, the triangle is deleted. It turned out that it is practical to also use $\mathrm{a}_{\max}$ as a threshold.

```
⟨SGNG topology: delete edges, triangles⟩ +≡                ← p. 90
  ⟨SGNG detect triangles covering holes, update $a_\triangle(\cdot)$  → p. 96⟩

  $\mathcal{F}_{\mathrm{a}_{\max}} = \{\triangle(\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k) \in \mathcal{F} \mid a_\triangle(\triangle(\mathbf{v}_i, \mathbf{v}_j, \mathbf{v}_k)) > \mathrm{a}_{\max}\}$

  $\mathcal{F} := \mathcal{F} \setminus \mathcal{F}_{\mathrm{a}_{\max}}$
```

Triangles covering holes

Once the edge and triangle creation step from the previous subsection has determined that an edge is required to construct the desired surface configuration, the triangles adjacent to this edge are checked if they are representing the input data. Let $(\mathbf{v}_l, \mathbf{v}_m)$ denote the required edge. Then, there are at most two triangles $\triangle(\mathbf{v}_l, \mathbf{v}_m, \mathbf{v}_n)$ and $\triangle(\mathbf{v}_l, \mathbf{v}_m, \mathbf{v}_o)$ adjacent to it. If both exist, the penalty of the triangle whose third vertex ($\mathbf{v}_n$ or $\mathbf{v}_o$) is closer to the current $\mathbf{p}_{\xi_t}$ is decremented by one, the penalty of the other is incremented by one. If only one of these triangles exists, its penalty is decremented. Decrementing is clamped so that $a_\triangle(\cdot) \geq 0$, denoted by the operator $\ominus$.

```
⟨SGNG detect triangles covering holes, update $a_\triangle(\cdot)$⟩ ≡        ← p. 95
  if $\mathcal{F}_e\big((\mathbf{v}_l, \mathbf{v}_m)\big) = \{\triangle(\mathbf{v}_l, \mathbf{v}_m, \mathbf{v}_n), \triangle(\mathbf{v}_l, \mathbf{v}_m, \mathbf{v}_o)\}$ then
    if $\|\mathbf{v}_n - \mathbf{p}_{\xi_t}\| \leq \|\mathbf{v}_o - \mathbf{p}_{\xi_t}\|$ then
      $a_\triangle\big(\triangle(\mathbf{v}_l, \mathbf{v}_m, \mathbf{v}_n)\big) := a_\triangle\big(\triangle(\mathbf{v}_l, \mathbf{v}_m, \mathbf{v}_n)\big) \ominus 1$
      $a_\triangle\big(\triangle(\mathbf{v}_l, \mathbf{v}_m, \mathbf{v}_o)\big) := a_\triangle\big(\triangle(\mathbf{v}_l, \mathbf{v}_m, \mathbf{v}_o)\big) + 1$
    else
      $a_\triangle\big(\triangle(\mathbf{v}_l, \mathbf{v}_m, \mathbf{v}_n)\big) := a_\triangle\big(\triangle(\mathbf{v}_l, \mathbf{v}_m, \mathbf{v}_n)\big) + 1$
      $a_\triangle\big(\triangle(\mathbf{v}_l, \mathbf{v}_m, \mathbf{v}_o)\big) := a_\triangle\big(\triangle(\mathbf{v}_l, \mathbf{v}_m, \mathbf{v}_o)\big) \ominus 1$
    end if
  else if $\mathcal{F}_e\big((\mathbf{v}_l, \mathbf{v}_m)\big) = \{\triangle(\mathbf{v}_l, \mathbf{v}_m, \mathbf{v}_n)\}$ then
    $a_\triangle\big(\triangle(\mathbf{v}_l, \mathbf{v}_m, \mathbf{v}_n)\big) := a_\triangle\big(\triangle(\mathbf{v}_l, \mathbf{v}_m, \mathbf{v}_n)\big) \ominus 1$
  end if
```

4.3.4 *Density Updates: Add, Delete Vertices*

During SGNG learning the density of the constructed mesh is adjusted in such a way that the reconstructed surface approximates the input points better. Vertices are added in regions with too much vertex activity $\tau(\cdot)$, whereas inactive vertices are removed. For this purpose each vertex keeps track of its activity τ, a value denoting how often the vertex has been selected best match:

$$\tau : \mathcal{V} \mapsto \mathbb{N} \quad , \text{ with } \tau(\mathbf{v}) := 0 \text{ initially} \quad .$$

The activity τ of a vertex is incremented by one, whenever it is selected best match for an input point.

Inactivity is addressed explicitly in SGNG, without changing the topology of the constructed mesh, i.e., cutting holes. Thus, vertices at protuberances, at inversions, or in regions that have been reconstructed too densely are removed while keeping the surface intact. For this purpose each vertex keeps track of the number ϑ of the last iteration it was selected as best match:

$$\vartheta : \mathcal{V} \mapsto \mathbb{N} \quad , \text{ with } \vartheta(\mathbf{v}) := t \text{ initially} \quad .$$

The value ϑ of the best matching vertex is updated in each iteration.

The density of the mesh is modified, whenever the number of input points processed so far is an integer multiple of a parameter λ. Activity and inactivity are addressed in the same iteration.

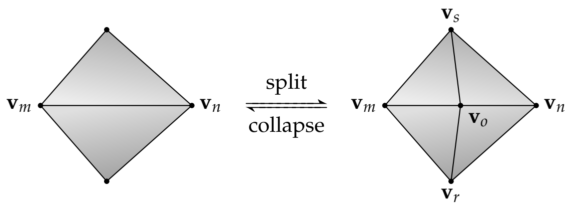


Figure 4.10: Splitting edge $(\mathbf{v}_m, \mathbf{v}_n)$, collapsing edge $(\mathbf{v}_o, \mathbf{v}_m)$.

```
⟨SGNG density updates⟩ ≡                                   ← p. 90
  $\tau(\mathbf{v}_b) := \tau(\mathbf{v}_b) + 1$
  $\vartheta(\mathbf{v}_b) := t$
  if $t \equiv 0 \pmod{\lambda}$ then
      ⟨SGNG mesh refinement  → p. 97⟩
      ⟨SGNG mesh coarsening  → p. 98⟩
  end if
```

SGNG mesh refinement

A new vertex $\mathbf{v}_o$ is added in regions with too much vertex activity. For this purpose the most active vertex of the triangle mesh is determined. The new vertex $\mathbf{v}_o$ is added in the middle of the longest edge $(\mathbf{v}_m, \mathbf{v}_n)$ emanating from the most active vertex $\mathbf{v}_m$. To preserve the constructed topology, the new vertex is created by splitting the edge $(\mathbf{v}_m, \mathbf{v}_n)$ and its adjacent triangles (Fig. 4.10 left to right). Activity of the three vertices $\mathbf{v}_m$, $\mathbf{v}_n$, $\mathbf{v}_o$ is set to the activity of the least active vertex of the mesh. Therefore, other regions with high activity will be handled before any of the three vertices becomes most active again. Thus, refinement is distributed globally across the triangle mesh.

```
⟨SGNG mesh refinement⟩ ≡                                   ← p. 97
  $\mathbf{v}_m = \arg\max_{\mathbf{v}_i \in \mathcal{V}} \tau(\mathbf{v}_i)$
  $\mathbf{v}_n = \arg\max_{\mathbf{v}_j \in \mathcal{N}_{\mathbf{v}_m}} \|\mathbf{v}_m - \mathbf{v}_j\|$
  $\mathcal{V} := \mathcal{V} \cup \{\mathbf{v}_o := \frac{\mathbf{v}_m + \mathbf{v}_n}{2}\}$
  split edge $(\mathbf{v}_m, \mathbf{v}_n)$ with new vertex $\mathbf{v}_o$
  $\tau(\mathbf{v}_m), \tau(\mathbf{v}_n), \tau(\mathbf{v}_o) := \min_{\mathbf{v}_k \in \mathcal{V} \setminus \{\mathbf{v}_o\}} \tau(\mathbf{v}_k)$
```

SGNG mesh coarsening

Vertices are removed in regions with too little vertex activity. For this purpose a set of inactive vertices is determined. A vertex is considered inactive, if it was not selected best match for a certain number $\Delta t_{\max} \cdot |\mathcal{V}|$ of iterations in a row. To preserve the constructed topology, an inactive vertex $\mathbf{v}_o$ is removed by collapsing one of the edges emanating from $\mathbf{v}_o$ (Fig. 4.10 right to left). For this purpose, SGNG determines the set $\mathcal{V}_m \subseteq \mathcal{N}_{\mathbf{v}_o}$ of target vertices that are directly connected to $\mathbf{v}_o$, onto which $\mathbf{v}_o$ may be collapsed while preserving the topological type of the constructed mesh. Hoppe et al. [65] defined criteria for this (Sec. 3.1). After that, SGNG picks that edge from the above set, for which an *edge collapse* maximizes regularity of the mesh,

Table 4.1: Learning parameters used in the SGNG examples.

Parameter		Value
Step size (best match $\mathbf{v}_\mathrm{b}$)	β	0.1
Step size (direct neighbors $\mathbf{v}_n \in \mathcal{N}_{\mathbf{v}_\mathrm{b}}$)	η	0.01
Maximum edge/triangle penalty	$\mathrm{a}_{\mathrm{max}}$	20
Iterations until density update	λ	100
Vertex inactivity threshold (used as $\Delta\mathrm{t}_{\mathrm{max}} \cdot \lvert\mathcal{V}\rvert$)	$\Delta\mathrm{t}_{\mathrm{max}}$	12

i.e., minimizes the sum of squared differences to valence six of the affected vertices:

$$E_c : \mathcal{E} \to \mathbb{R}$$
$$\text{with } E_c\big((\mathbf{v}_o, \mathbf{v}_m)\big) = \left(\lvert\mathcal{N}_{\mathbf{v}_m}\rvert + \lvert\mathcal{N}_{\mathbf{v}_o}\rvert - \lvert\mathcal{N}_{\mathbf{v}_m} \cap \mathcal{N}_{\mathbf{v}_o}\rvert - 8\right)^2 + \sum_{\mathbf{v}_k \in \mathcal{N}_{\mathbf{v}_m} \cap \mathcal{N}_{\mathbf{v}_o}} \left(\lvert\mathcal{N}_{\mathbf{v}_k}\rvert - 7\right)^2 \quad ,$$

where $\lvert\mathcal{N}_{\mathbf{v}_k}\rvert$ denotes the number of neighbors of a vertex $\mathbf{v}_k$, $\mathbf{v}_o$ denotes the inactive vertex, and $\mathbf{v}_m$ denotes the other endpoint of the selected edge (Fig. 4.10 right).

```
⟨SGNG mesh coarsening⟩ ≡                                   ← p. 97
  $\mathcal{V}_o = \{\mathbf{v}_o \in \mathcal{V} \mid \vartheta(\mathbf{v}_o) < t - \Delta\mathrm{t}_{\mathrm{max}} \cdot \lvert\mathcal{V}\rvert\}$
  for each $\mathbf{v}_o \in \mathcal{V}_o$ do
      $\mathcal{V}_o^+ = \{\mathbf{v}_l \in \mathcal{N}_{\mathbf{v}_o} \mid$ collapsing $(\mathbf{v}_o, \mathbf{v}_l)$ is valid [65]$\}$
      if $\lvert\mathcal{V}_o^+\rvert > 0$ then
          $\mathbf{v}_m = \arg\min_{\mathbf{v}_l \in \mathcal{V}_o^+} E_c\big((\mathbf{v}_o, \mathbf{v}_l)\big)$
          collapse edge $(\mathbf{v}_o, \mathbf{v}_m)$ removing vertex $\mathbf{v}_o$
          $\mathcal{V} := \mathcal{V} \setminus \{\mathbf{v}_o\}$
      end if
  end for each
```

SGNG keeps the reconstructed surface intact as long as possible: If a vertex becomes inactive, because it has been moved into a hole that is present in the input data, SGNG will first remove the vertex with the above technique. Topology updates in later iterations will detect that the resulting triangles are covering a hole and remove them.

4.4 EXAMPLES FOR SGNG LEARNING

In order to illustrate the characteristics of the proposed SGNG and to show the difference to the existing algorithms that are described in Ch. 2, examples are presented in this subsection. The same data sets are used as before to make the results comparable. Fig. 4.5 shows an example of an initial SGNG configuration. The network initially contains two randomly placed vertices. The learning parameters that are used in these reconstruction examples are collected in Tab. 4.1.

These values have been determined in a separate experiment that is described later (Sec. 4.6).

Sgng exports only those edges and vertices as the final constructed mesh that have adjacent triangles (Sec. 4.2). Additional edges and vertices with no adjacent triangles are needed in early iterations to let sgng explore the data. Such edges and vertices are denoted by dotted lines (⋯⋯) and white-filled circles (○), respectively, in Fig. 4.11, 4.12.

Sgng reconstructing a square

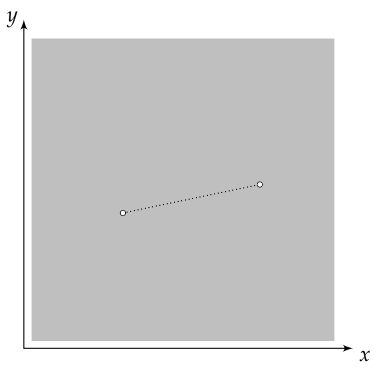
In the first example the input points $\mathcal{P} \subset \mathcal{P}_{\square}$ (Eq. 2.1) are drawn uniformly at random from a square. Fig. 4.11 shows the resulting mesh after different numbers of iterations. At first, the results are very similar to those of gng. In the first iteration both initial vertices get connected by an edge. During the first 99 iterations this line segment moves and gets longer to match the input points better (Fig. 4.11(a)). It is split and it bends once $t \geq \lambda$ (Fig. 4.11(b)). The vertices get connected to a first three-loop during the next few iterations, and a triangular face is added to the mesh (Fig. 4.11(c)). Afterwards, the constructed mesh is refined repeatedly. The vertices spread out, new edges are added and obsolete ones are deleted (Fig. 4.11(d)–(g)). In contrast to gsrm edge splits do not remove any desired triangles and thus do not leave untriangulated n-loops of edges. Eventually, the vertices are distributed fairly evenly across the input square (Fig. 4.11(h)). The constructed triangles have very similar sizes with only few irregularities. Almost no triangles are being degenerate. Furthermore, sgng creates all desired triangles during learning. Thus, post-processing steps that are inevitable for gsrm are obsolete with sgng.

Sgng reconstructing an annulus

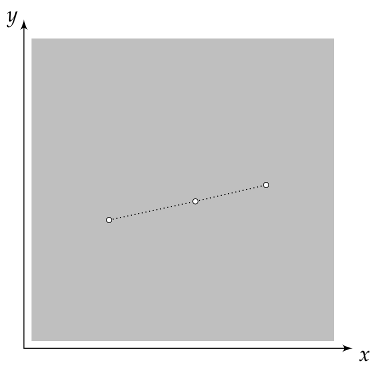
In the second example the input points $\mathcal{P} \subset \mathcal{P}_{\circledcirc}$ (Eq. 2.2) are drawn uniformly at random from an annulus. Fig. 4.12 shows the resulting mesh after different numbers of iterations. The characteristics of the results are very similar to the square example. At first, a triangulated quadrangle is created (Fig. 4.12(d)). In the early iterations, the density and distribution of the constructed vertices do not match the density and distribution of the input points, leading to holes and overlapping edges (Fig. 4.12(e)). However, sgng recovers easily from this configuration during the next few iterations constructing a triangulated disk (Fig. 4.12(f)) and afterwards an annulus (Fig. 4.12(g)). Finally, all constructed vertices are fairly evenly distributed across the input annulus with the hole being correctly reconstructed (Fig. 4.12(h)). As in the previous example the constructed triangles have very similar sizes with only few irregularities. Almost no triangles are being degenerate. Furthermore, sgng creates all desired triangles during learning without requiring any post-processing steps.

4.5 implementation details

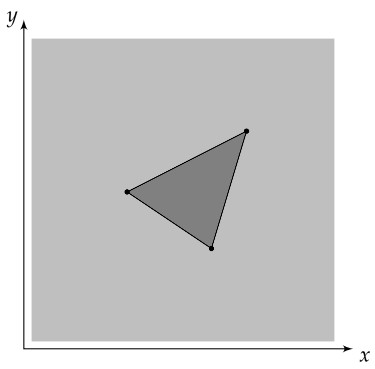
During implementation of sgng, some care must be taken to achieve runtime efficiency. For the sake of clarity, the sgng algorithm pre-



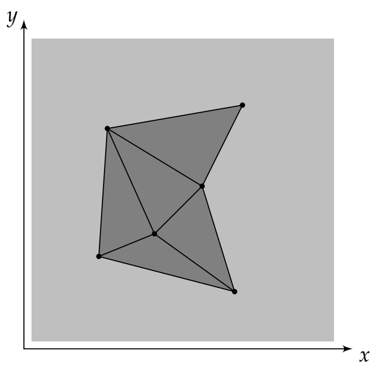
(a) $t = 99$ iterations



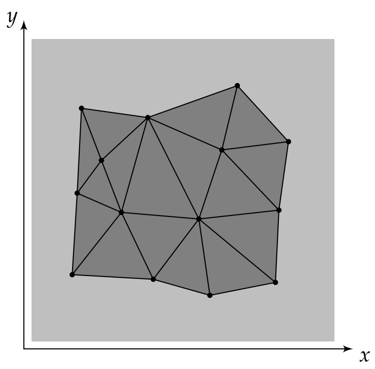
(b) $t = 110$ iterations



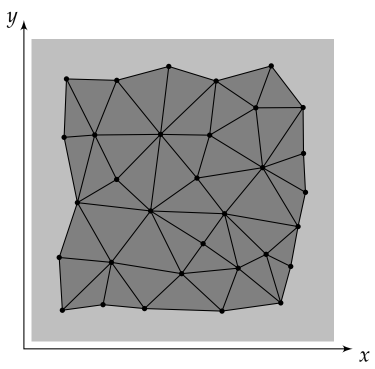
(c) $t = 199$ iterations



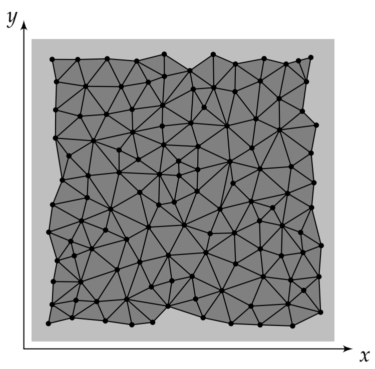
(d) $t = 480$ iterations



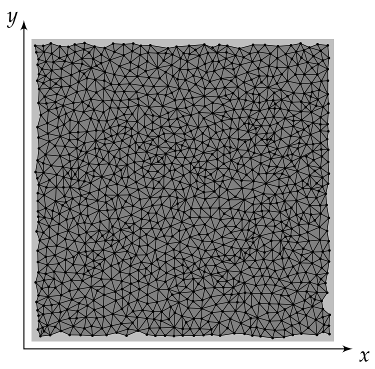
(e) $t = 1200$ iterations



(f) $t = 3000$ iterations

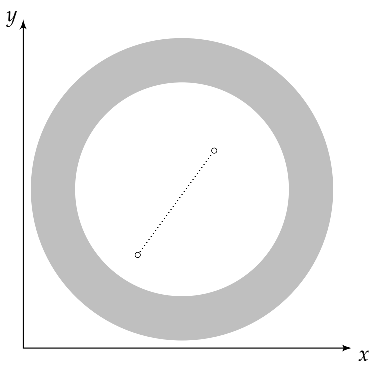


(g) $t = 12\,000$ iterations

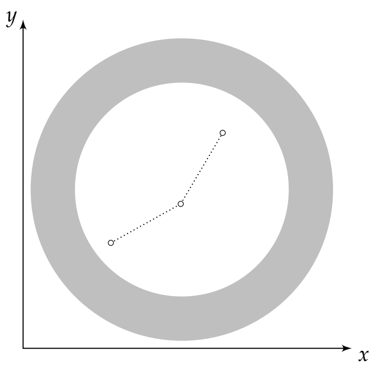


(h) $t = 120\,000$ iterations

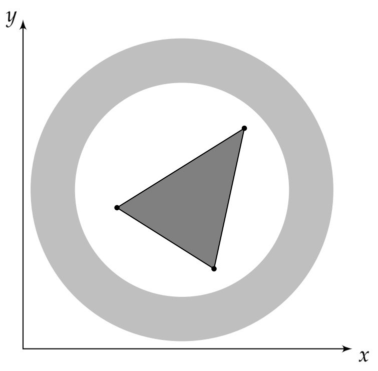
Figure 4.11: SGNG reconstructing a square.



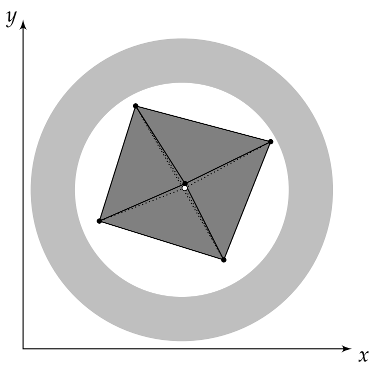
(a) $t = 99$ iterations



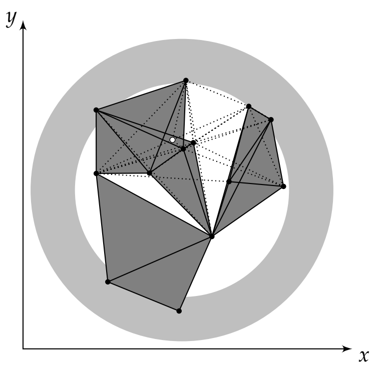
(b) $t = 110$ iterations



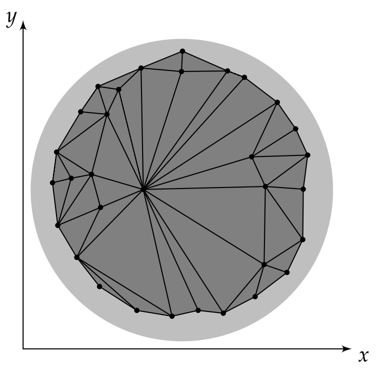
(c) $t = 199$ iterations



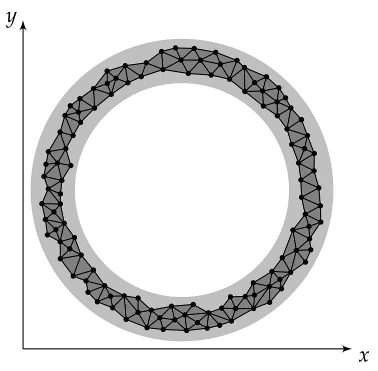
(d) $t = 480$ iterations



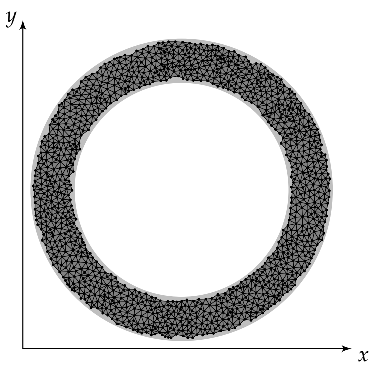
(e) $t = 1200$ iterations



(f) $t = 3000$ iterations



(g) $t = 12\,000$ iterations



(h) $t = 120\,000$ iterations

Figure 4.12: SGNG reconstructing an annulus.

sented so far was described without referring to implementation details. However, many design decisions have been made with an efficient implementation in mind. To make the results presented in this dissertation reproducible, important details about the implementation are outlined in this section. A pseudocode that is very similar to C++ is used for this purpose.

Custom data structures for the constructed mesh, its triangles, its edges, and its vertices have been designed for SGNG in order to support the required operations with low overhead.

```
⟨SGNG data structures⟩ ≡
  ⟨SGNG Mesh class  → pp. 102, 103⟩
  ⟨SGNG Vertex class  → pp. 103, 104⟩
  ⟨SGNG Edge class  → p. 102⟩
  ⟨SGNG Triangle class  → p. 102⟩
  ⟨SGNG OctreeNode class  → p. 103⟩
```

Mesh class The constructed mesh maintains lists of vertices, edges, and triangles. In order to handle activity and inactivity of vertices, two separate vertex lists are required. Details are presented later.

```
⟨SGNG Mesh class⟩ ≡                                      ← p. 102
  Vertex* pFirstActiveVertex;     // $\mathcal{V}$, Activity list
  Vertex* pFirstInactiveVertex;   // $\mathcal{V}$, Inactivity list
  list<Edge*> pEdges;             // $\mathcal{E}$
  list<Triangle*> pTriangles;     // $\mathcal{F}$
```

Triangle class Each triangle stores pointers to the three adjacent edges. The vertices and adjacent triangles are accessible via these edges. In order to detect and remove obsolete triangles, each triangle stores the penalty assigned to it.

```
⟨SGNG Triangle class⟩ ≡                                  ← p. 102
  Edge* pAdjEdges[3];  // $(\mathbf{v}_0,\mathbf{v}_1),(\mathbf{v}_1,\mathbf{v}_2),(\mathbf{v}_2,\mathbf{v}_0)$
  int*  penalty;       // $a_\triangle(\triangle(\mathbf{v}_0,\mathbf{v}_1,\mathbf{v}_2))$
```

Edge class Each edge stores pointers to the two vertices that it is connecting, and pointers to at most two adjacent triangles. Finally, each edge keeps track of its penalty in order to detect and remove obsolete edges.

```
⟨SGNG Edge class⟩ ≡                                      ← p. 102
  Vertex* pVertices[2];          // $\mathbf{v}_0,\mathbf{v}_1$
  Triangle* pAdjTriangles[2];
  int penalty;                   // $a_e((\mathbf{v}_0,\mathbf{v}_1))$
```

Vertex class Besides its 3D position each vertex stores a list of pointers to the edges emanating from it. Thus, SGNG efficiently checks if two given vertices are already connected by an edge. Furthermore, SGNG efficiently determines the set of neighbors of a single given vertex and the set of common neighbors of two given vertices. SGNG also efficiently checks whether a given triangle already exists. Finally, each vertex

$\mathbf{v}$ keeps track of its activity $\tau(\mathbf{v})$ and the number $\vartheta(\mathbf{v})$ of the last iteration in which it was selected as best match.

⟨*SGNG* Vertex *class*⟩ ≡ ← p. 102

```
  float position[3];      // v
  list<Edge*> pEdges[2];
  int activity;           // τ(v)
  int lastIterBestMatch;  // ϑ(v)
```

With the above data structures both *edge split* and *edge collapse* are implemented efficiently. The vertices that are affected by both operations are found via the set of common neighbors of the vertices that are connected by the respective edge. Furthermore, the data structures allow for efficient neighbor search and efficient detection of undesired edges and triangles. The set of edges that are emanating from the best matching vertex is used to find the neighbors of the best matching vertex that are affected by position updates. The same set of emanating edges is used when checking for undesired edges. Only the triangles adjacent to the edges emanating from the best matching vertex are considered when checking for undesired triangles.

Vertex positions are stored in the leaf nodes of an octree that is dynamically updated during reconstruction. That way the best and the second-best matching vertex for a randomly selected input point are found efficiently. $\mathbf{v}_b, \mathbf{v}_c$

⟨*SGNG* Mesh *class*⟩ +≡ ← p. 102

```
  OctreeNode* pOctreeRoot;
```

A leaf node contains at most eight vertices. Nodes are split when necessary, and collapsed when possible. The depth of the octree is not limited. Whenever a vertex is moved beyond or a new vertex is placed outside the bounds of the root node, the octree grows upwards by creating a new root node having the old root node as one of its child nodes.

⟨*SGNG* OctreeNode *class*⟩ ≡ ← p. 102

```
  OctreeNode* pParent;
  float bounds[6];
  bool isLeaf;
  union {
    OctreeNode* pChildNodes[8];
    Vertex* pVertices[8];  }
```

To facilitate the dynamic updates each vertex stores a pointer to the octree node it is currently located in. That way, SGNG efficiently checks whether a vertex has been moved beyond the bounds of the respective node and whether parts of the octree have to be updated.

⟨*SGNG* Vertex *class*⟩ +≡ ← p. 102

```
  OctreeNode* pOctreeNode;
```

Vertex activity and inactivity

SGNG organizes the vertices in two lists: The entries in the activity list are sorted with respect to the activity of the corresponding vertices, and the entries in the inactivity list are sorted with respect to the number of the last iteration in which the corresponding vertices were created or selected as best match. Thus, the most active vertex is located at the front of the first list, and inactive vertices are located at the end of the second list.

Whenever a vertex is selected as best match, its activity is incremented by one, and its position in the activity list is readjusted. Simultaneously, it is moved to the front of the inactivity list. A density update resets the activity of the affected vertices to the smallest activity value in the mesh. The vertices are thus moved to the end of the activity list. Each vertex stores pointers to its immediate predecessor and successor in both lists. That way, separate list nodes do not need to be allocated, and thus memory requirements are reduced.

$\langle$*SGNG* `Vertex` *class*$\rangle$ $+\equiv$ ← p. 102

```
Vertex* pPrevVtxInActivityList;
Vertex* pNextVtxInActivityList;
Vertex* pPrevVtxInInactivityList;
Vertex* pNextVtxInInactivityList;
```

4.6 EXPERIMENTS AND RESULTS

At first, suitable reconstruction parameters for SGNG were determined that were then used in all subsequent experiments (Tab. 4.1). These parameters were also used in the examples presented earlier. Afterwards, experiments were conducted in order to illustrate the effect of the new boundary fitting and to demonstrate that SGNG can handle input data that is locally homogeneous but that has a globally varying density. In order to verify the performance of SGNG, and to compare it to other surface reconstruction techniques, SGNG was tested with the benchmark provided by Berger et al. [13, 14]. By repeating their error distribution experiments for *Poisson surface reconstruction* (PSR) the results for SGNG can be related to the results of other state-of-the-art reconstruction techniques that were tested in the original benchmark.

SGNG is compared to GSRM in order to verify whether the new algorithm does indeed improve the reconstruction process. Furthermore, SGNG is compared to *screened Poisson surface reconstruction* (SPSR) [78] as a state-of-the-art baseline. This comparison focuses mainly on iterative reconstructions providing preview while reconstructing from real-world data. At present SPSR is designed to construct only watertight meshes. However, its authors provide a trimming tool to remove undesired triangles. To make comparison fair, this trimming tool was used in the experiments. In addition to creating high-resolution renderings of the reconstructed surfaces for visual comparison, the median reconstruction time of ten executions was determined in the experiments. Additionally the mean relative error normalized to the length of the diagonal of the bounding box was evaluated. Finally, the angle distribution and the reconstruction quality were computed.

SGNG was implemented in C++ and executed as a 64 bit-application on a single CPU core. It was not especially fine-tuned for run-time performance. However, a dynamic octree was used to determine $\mathbf{v}_b$ and $\mathbf{v}_c$. GSRM was implemented and configured according to the original paper [41], triangulating 3- to 7-loops during post-processing. The dynamic octree and the new activity updates of SGNG were used to make comparison fair. PSR version 2 [76] as provided with the benchmark, and the recent SPSR version 6.13 [77] were used. Experiments were run on an Apple® MacBook Pro® with a 2.6 GHz Intel® Core™ i7-4960HQ CPU, 16 GB 1600 MHz DDR3 RAM, running Mac OS X® 10.9.

Reconstruction error

The mean distance between the input points and the constructed triangles was computed as an error measure. That way, the overall deviation of the output mesh from the input points is tracked better than using the Hausdorff distance. The latter yields only the largest deviation and is thus prone to outliers.

The *metro tool* [27] could not be used directly due to data incompatibilities. Thus, the measurement process was rebuilt: *Monte Carlo sampling* from MeshLab [28] was used to create a set $\mathcal{S}$ of 3D samples lying on the triangles of each exported mesh. The number $|\mathcal{S}|$ of samples was equal to the number $|\mathcal{P}|$ of input points used to construct the respective mesh. Afterwards, two mean distance values were computed: The first indicating the mean distance from an input point $\mathbf{p}_i \in \mathcal{P}$ to the closest sample $\mathbf{s}_j \in \mathcal{S}$, the second indicating the mean distance from a sample to the closest input point. To be comparable throughout the experiments, both values were normalized to the length of the diagonal of the bounding box of the point cloud, $d_{\mathcal{P}}$, or of the set of samples, $d_{\mathcal{S}}$, respectively. The greater value of the two mean distance values was selected as the error value e:

$$e = \max\left\{ \frac{1}{|\mathcal{P}|} \sum_{i=1}^{|\mathcal{P}|} \min_{\mathbf{s}_j \in \mathcal{S}} \frac{\|\mathbf{p}_i - \mathbf{s}_j\|}{d_{\mathcal{P}}}, \frac{1}{|\mathcal{S}|} \sum_{j=1}^{|\mathcal{S}|} \min_{\mathbf{p}_i \in \mathcal{P}} \frac{\|\mathbf{p}_i - \mathbf{s}_j\|}{d_{\mathcal{S}}} \right\} \quad .$$

Reconstruction quality

Reconstruction quality was evaluated in terms of triangle shape: The more regular, i.e., the closer to equilateral the triangles, the better. Therefore, the ratio of the radii of the incircle and the circumcircle of a triangle was used as a quality measure, normalized to the ratio of an equilateral triangle.

Let a, b, c denote the lengths of the sides of a triangle, then for the area A of the triangle [125],

$$A = \frac{1}{4}\sqrt{(a+b+c)(b+c-a)(c+a-b)(a+b-c)} \quad ,$$

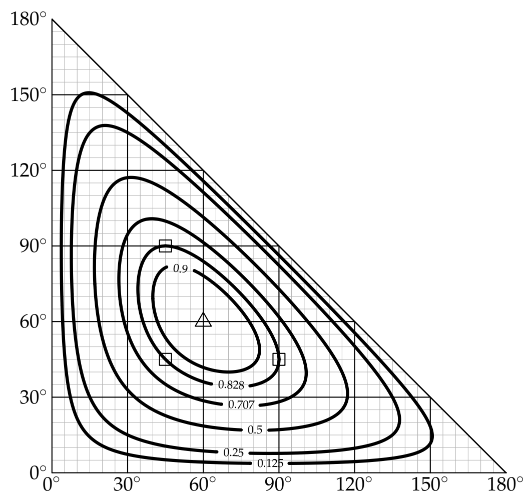


Figure 4.13: Contour plot of reconstruction quality values for all valid triangle configurations. The abscissa and ordinate specify two of the three inner angles. △ marks the quality value ($q = 1.0$) of equilateral tringles, and □ marks the quality value ($q \approx 0.828$) of half-square triangles.

Table 4.2: Characteristic quality values

Type of triangle	Quality (q)
Equilateral	1.0
Half-square	≈0.828
Isosceles, top angle ≈ 34.1°	≈0.828
Isosceles, top angle < 26.5° or > 100.8°	<0.707
Isosceles, top angle < 16.9° or > 117.2°	<0.5

its inradius r, and its circumradius R [126, 127]

$$r = \frac{2A}{a+b+c} \quad , \quad R = \frac{abc}{4A} \quad ,$$

and thus

$$q = \frac{2r}{R} = \frac{16A^2}{(a+b+c) \cdot abc} \in [0,1] \quad ,$$

where the factor of 2 is used for normalization so that $q = 1$ for equilateral triangles. The contour plot in Fig. 4.13 gives an overview of the quality values for different angle configurations in a triangle. Tab. 4.2 lists characteristic quality values for some triangle types.

Determining suitable parameter values

In order to determine a set of suitable parameter values for the experiments, SGNG reconstructions that were created with various combinations of parameter values were evaluated. To make sure that the subsequent experiments yielded results that are related to the generalization capabilities of SGNG, a different data set was used here: A set of $200 \cdot 10^3$ points sampled from the Armadillo reconstructions

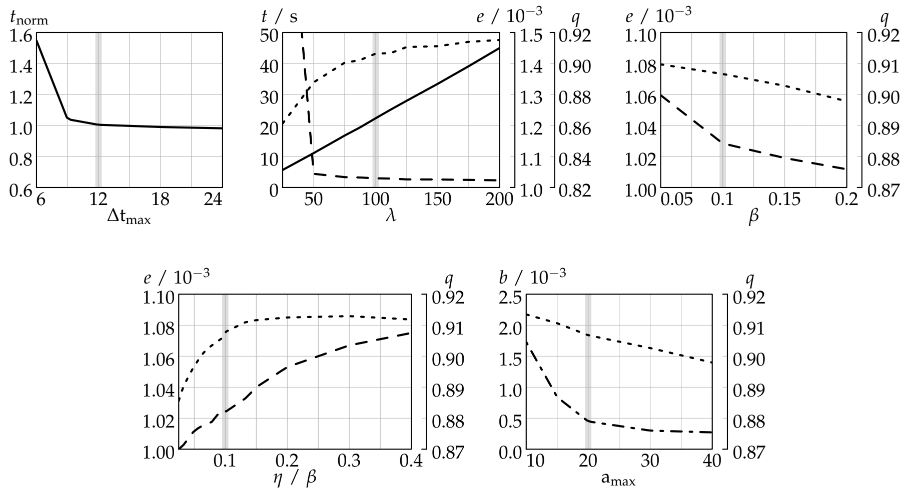


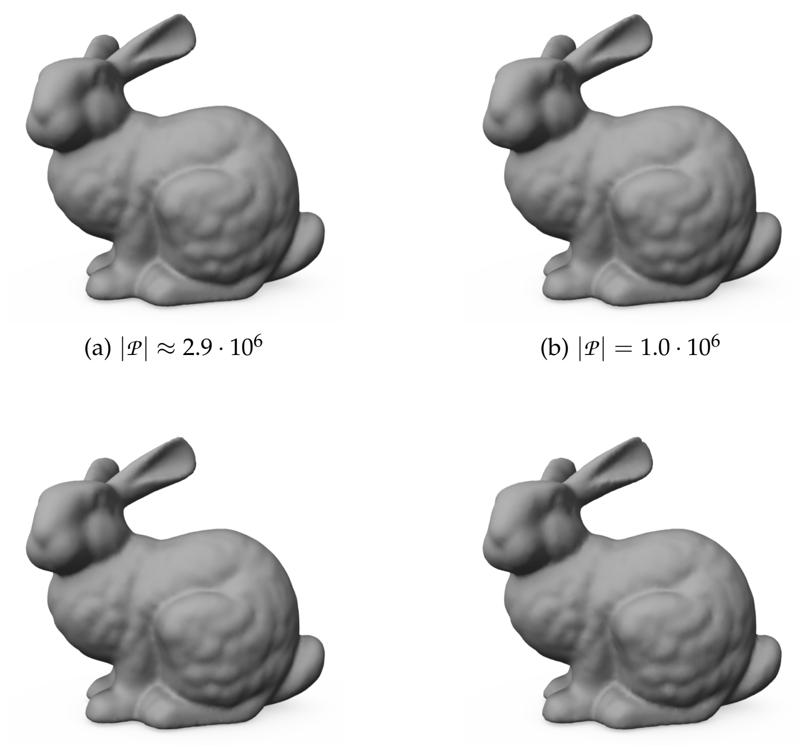
Figure 4.14: Influence of the learning parameters of SGNG on (normalized) reconstruction time (t (t_{norm}), solid), error (e, dashed), quality (q, dotted), and relative number of boundary edges (b, dash-dotted). The parameter values used in the experiments are highlighted by a vertical gray line.

from Stanford Scanning Repository [114]. SGNG constructed meshes with $50 \cdot 10^3$ vertices.

Fig. 4.14 shows plots revealing the relationships between the learning parameters and reconstruction time t (solid), error e (dashed), and quality q (dotted). The leftmost plot uses t_{norm}, the time normalized to the result at $\Delta t_{\max} = 12$ for each tested value of λ, to account for the proportionality relationship of t and λ. Since a point to vertex ratio of 4 was used here, SGNG leaves a small number of holes untriangulated in regions where the vertex density exceeds the point density. The number of holes is represented by the relative number b of boundary edges in the lower right plot. From the plots a set of suitable parameter values ($\beta = 0.1$, $\eta = 0.01$, $\lambda = 100$, $\Delta t_{\max} = 12$, $a_{\max} = 20$) (Tab. 4.1) was determined as trade-offs between speed, quality, error, and the number of erroneously untriangulated holes. A vertical gray line highlights these values in the plots.

Comparison to GSRM

The experiment that was conducted earlier in order to demonstrate the influence of the input data's density to GSRM reconstructions (Fig. 4.2) was repeated for SGNG. Fig. 4.15 shows renderings of SGNG reconstructions of Stanford Bunny with $20 \cdot 10^3$ vertices from different numbers of input points. While the GSRM reconstructions contain holes for sparser input data, the new topology learning (Sec. 4.3.2, 4.3.3) enables SGNG to construct a hole-free triangle mesh even from sparse input data that can be used directly for visualization at any time during learning. In addition to that, SGNG is faster than GSRM (Tab. 4.3). While the post-processing of GSRM took longer the sparser the input data was, running times of SGNG are almost constant independent of the number of input points. For fairness, $\lambda = 200$ for both GSRM



(a) $|\mathcal{P}| \approx 2.9 \cdot 10^6$ (b) $|\mathcal{P}| = 1.0 \cdot 10^6$

(c) $|\mathcal{P}| = 0.3 \cdot 10^6$ (d) $|\mathcal{P}| = 0.1 \cdot 10^6$

Figure 4.15: SGNG reconstructions of Stanford Bunny with $20 \cdot 10^3$ vertices from different numbers of input points. Results for GSRM reconstructions of same sets of input points are presented in Fig. 4.2.

and SGNG in this experiment, since this value was used in the original paper by do Rêgo et al. [41].

Effect of additional boundary fitting

In order to evaluate the effect of the newly introduced boundary fitting (Sec. 4.3) a set of 12 000 input points was used that were sampled from a square with an area of 1 square unit. From these input points SGNG constructed two meshes containing 100 vertices each: the first without, the second with the additional boundary fitting. Fig. 4.16 presents the constructed meshes that both approximate the original square. The input points are overlaid for reference. Without boundary fitting (Fig. 4.16(a)) SGNG constructs a mesh with a surface area of approximately 0.72 square units. With boundary fitting (Fig. 4.16(b)) SGNG constructs a mesh with a surface area of approximately 0.81 square units. Therefore, the newly introduced boundary fitting reduces the error in terms of surface area and thus improves boundary accuracy—by 32 % in this experiment.

Varying point density

An experiment was conducted in order to demonstrate that SGNG can handle input data robustly even if its density is locally homogeneous but varies across the original surface. For this purpose a torus was sampled with four different densities (Fig. 4.17(a)) yielding

Table 4.3: Running times in seconds for gsrm and sgng reconstructions of Stanford Bunny with $|\mathcal{V}| = 20 \cdot 10^3$.

	Gsrm			Sgng		
$\lvert\mathcal{P}\rvert$ / 10^6	2.9	0.3	0.1	2.9	0.3	0.1
Reconstruction	9.7	9.3	8.7	15.5	15.2	15.1
Post-Processing	8.9	17.3	43.0	–	–	–
Total	18.6	26.6	51.7	15.5	15.2	15.1

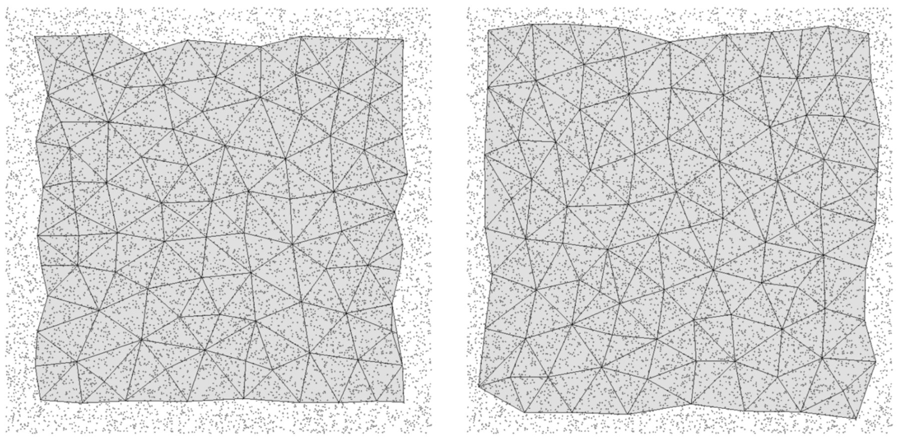

(a) Without boundary fitting. (b) With boundary fitting.

Figure 4.16: Effect of additional boundary fitting: Sgng reconstructions of a square with overlaid input points.

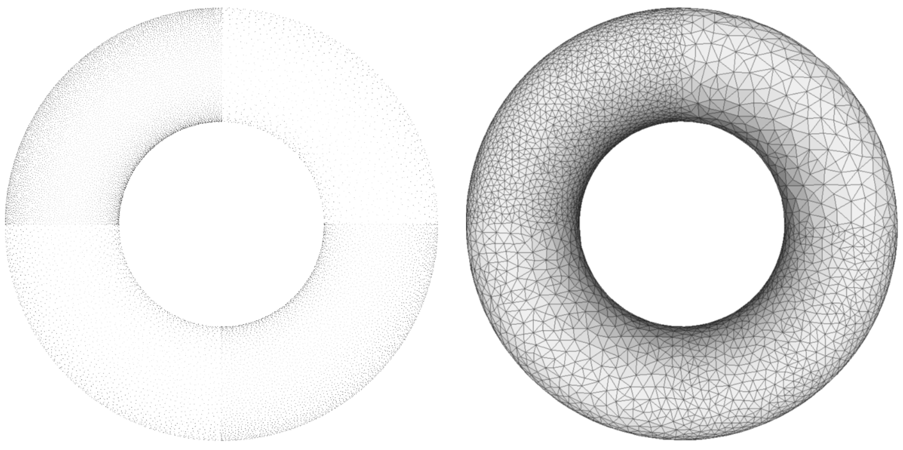

(a) (b)

Figure 4.17: Torus with varying point density. (a) Input points: Upper right 1479 points, lower right 5878 points, lower left 2955 points, upper left 11 723 points. (b) Sgng reconstruction with overlaid wireframe.

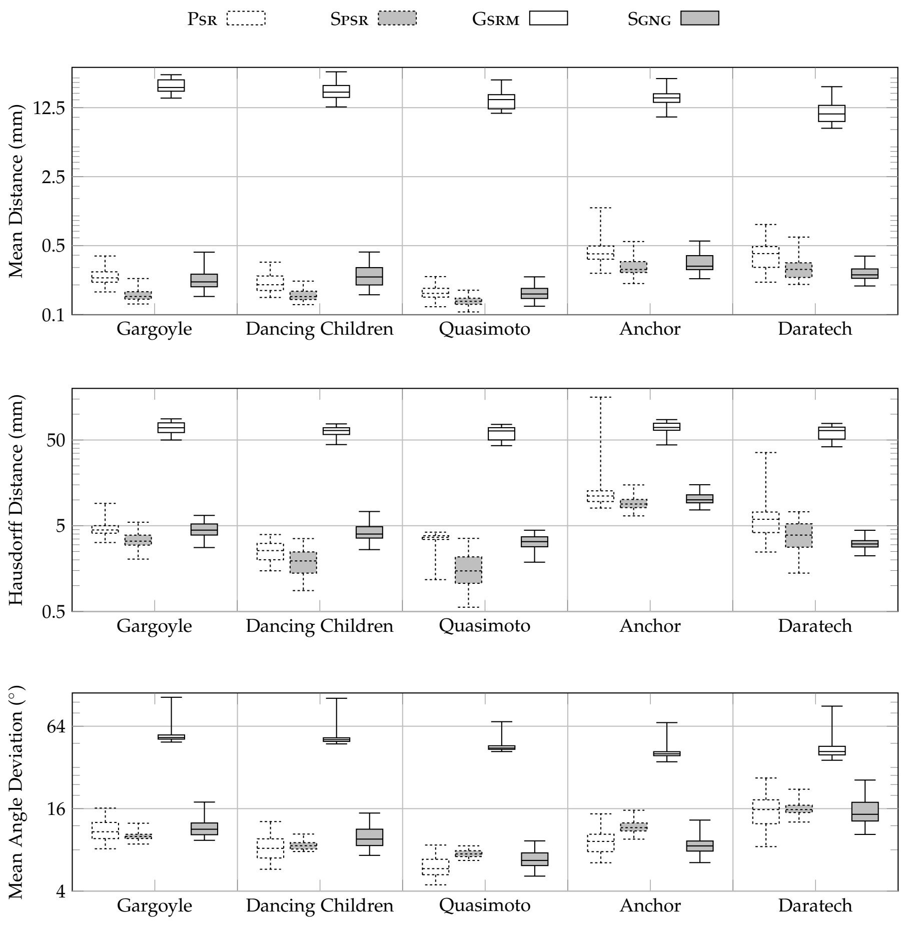


Figure 4.18: Plots for the error distribution experiments using the benchmark test provided by [14].

a total of 22 035 input points. Using the above learning parameters SGNG constructed a smooth surface with 5508 vertices from these input points that approximates the original torus (Fig. 4.17(b)). Due to the new topology learning (Sec. 4.3.2, 4.3.3) and the improved density learning (Sec. 4.3.4) SGNG is robust to the varying density of the input points: Constructed triangles are smaller in regions with greater point density than in regions with lower density. Since even in the sparsely sampled regions the triangles are large compared to the distance of the input points, there are no untriangulated holes in the constructed surface. If the settings of SGNG were changed in such a way that more and thus smaller triangles were constructed, SGNG would leave some holes untriangulated. However, this could be detected easily during iterative reconstruction, and an operator could adjust the parameters accordingly without having to restart reconstruction from scratch.

Benchmark tests In order to compare SGNG to other surface reconstruction techniques, SGNG was evaluated using the error distribution tests of a publicly available benchmark [14]. The test for PSR was repeated as a verification: The results that were published by Berger et al. [14] could

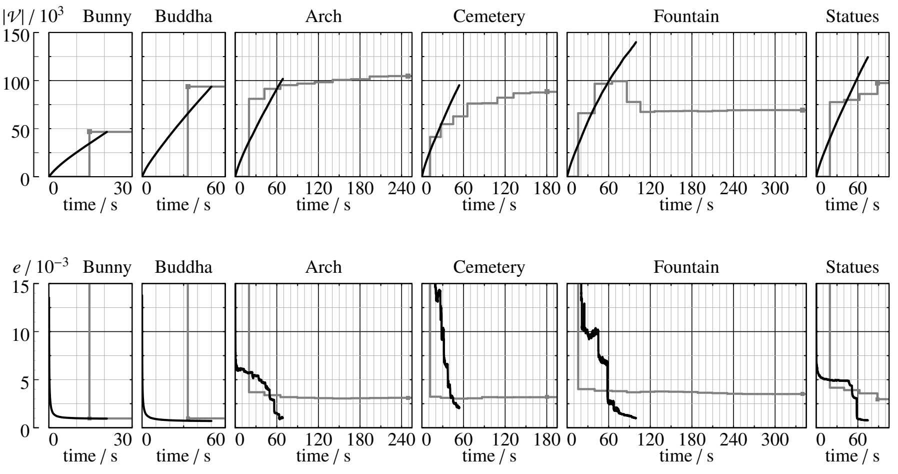


Figure 4.19: Number $|\mathcal{V}|$ of vertices (top row) and mean error e (bottom row) of sgng (black) and spsr (gray) reconstructions versus median time.

be reproduced. The results of psr thus serve as a link to the results of the other techniques that Berger et al. tested. In addition to gsrm, spsr was also tested, since sgng is compared to it. Fig. 4.18 presents the results. Spsr performed slightly better than psr in terms of mean and Hausdorff distance, but led to slightly larger angular errors. Since the point clouds that were used in the benchmark are too sparse for hole-free gsrm reconstructions, these caused large errors.

Sgng yielded results that are comparable to those presented by Berger et al.: In terms of mean distance, Hausdorff distance, and mean angle deviation for Gargoyle, Dancing Children, and Quasimoto sgng performed similar to psr, and thus similar to or better than half of the algorithms that were tested in the original benchmark [14]. For the Anchor and Daratech data sets performance of sgng was even better: Sgng outperformed psr in terms of mean and Hausdorff distance, with similar results as spsr.

In the original benchmark psr reconstruction took 36.83 s, averaged across all point clouds [14]. In the experiment that was repeated here psr reconstruction took 36.91 s. Thus, running times can be compared directly to the results of the other techniques that were tested in the original benchmark. For comparison, in this experiment, spsr reconstruction with the same settings as psr took 130.79 s, gsrm reconstruction took 103.68 s including post-processing, and sgng reconstruction took 19.66 s.

Real-world reconstruction experiments

Sgng is compared to spsr using six real-world data sets: The original range data of Stanford Bunny and Happy Buddha from Stanford Scanning Repository [114], and the Arch, Cemetery, Fountain, and Statues data sets created by a combination of *VisualSFM* [130] and *CMVS/PMVS* [54] from photos taken in Paris, France. Stanford Bunny and Happy Buddha are used for single-shot reconstructions. The

Table 4.4: Results for SGNG and SPSR reconstructions.

	Bunny		Buddha	
	SGNG	SPSR	SGNG	SPSR
Points (iterative steps)	362 230 (1 St.)		1 098 870 (1 St.)	
Vertices	46 634		93 732	
Time / s	20.9	14.6	50.0	33.0
Mean rel. error / 10^{-3}	0.9	1.0	0.7	1.0

	Arch		Cemetery	
	SGNG	SPSR	SGNG	SPSR
Points (iterative steps)	407 229 (10 St.)		380 840 (9 St.)	
Vertices	101 808	104 670	95 210	88 303
Time / s	69.2	249.2	53.8	180.3
Mean rel. error / 10^{-3}	1.0	3.1	2.1	3.2

	Fountain		Statues	
	SGNG	SPSR	SGNG	SPSR
Points (iterative steps)	567 920 (16 St.)		497 489 (4 St.)	
Vertices	140 000	69 248	124 373	97 408
Time / s	99.4	339.8	74.5	88.2
Mean rel. error / 10^{-3}	1.0	3.5	0.8	3.0

remaining data sets are used in experiments that constitute a mock-up of the intended application of SGNG: An interactive pipeline where data acquisition and reconstruction with continuous preview are executed in parallel. For this purpose incremental input point clouds are constructed by adding only one photo at a time. To determine the net reconstruction time in such a pipeline, the experiments start with the initial point cloud. Further input points for the next incremental step are added once a point to vertex ratio of 4 is reached (SGNG) or the previous reconstruction task has been finished (SPSR). SGNG is configured to use the previously determined learning parameters. SPSR uses a 128^3 voxel grid for Bunny and a 256^3 voxel grid for the other data sets. For Buddha a trimming value of 9 is used, for the other data sets a value of 7 is used.

Fig. 4.19 and Tab. 4.4 present the results of these experiments. SPSR is faster than SGNG by a factor of approximately 1.5 for single-shot reconstructions of Bunny and Buddha. However, it can be seen in the plots in Fig. 4.19 that SGNG instantly provides an approximate reconstruction that gets continuously refined. For the single-shot reconstructions of Bunny and Buddha the error of SGNG is slightly lower than the error of SPSR. The former gets reduced during learning with a steep initial decay. For iterative reconstructions of Arch, Cemetery, and Fountain that are updated frequently SGNG is faster than SPSR by a factor of 3.4 to 3.6 and still by a factor of 1.1 for the Statues data set with only 3 updates during reconstruction. Again, the error of SGNG is considerably smaller than the error of spsr. The former gets reduced during learning with a steep initial decay, but increases temporarily every time new points are added.

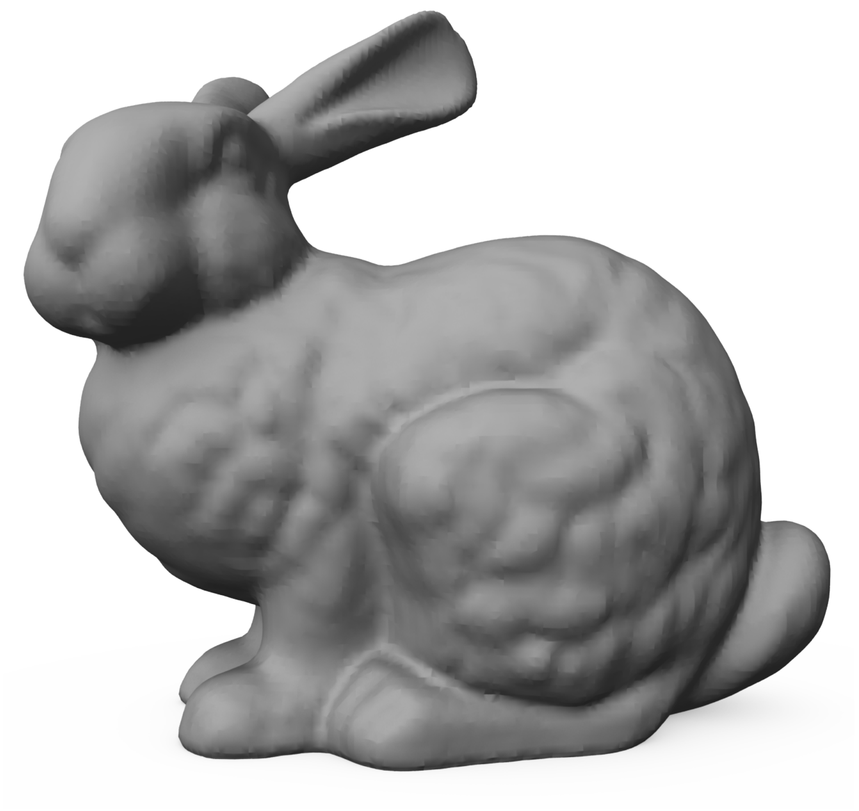
Visual evaluation

Besides the above measurements the visual appearance of the reconstructions is evaluated. Fig. 4.20–4.32 present the results. For each iterative reconstruction a photo is presented as an overview (Fig. 4.24(a), 4.26(a), 4.28(a), 4.30(a)) as well as the complete point cloud (Fig. 4.24(b), 4.26(b), 4.28(b), 4.30(b)) created from the input images.

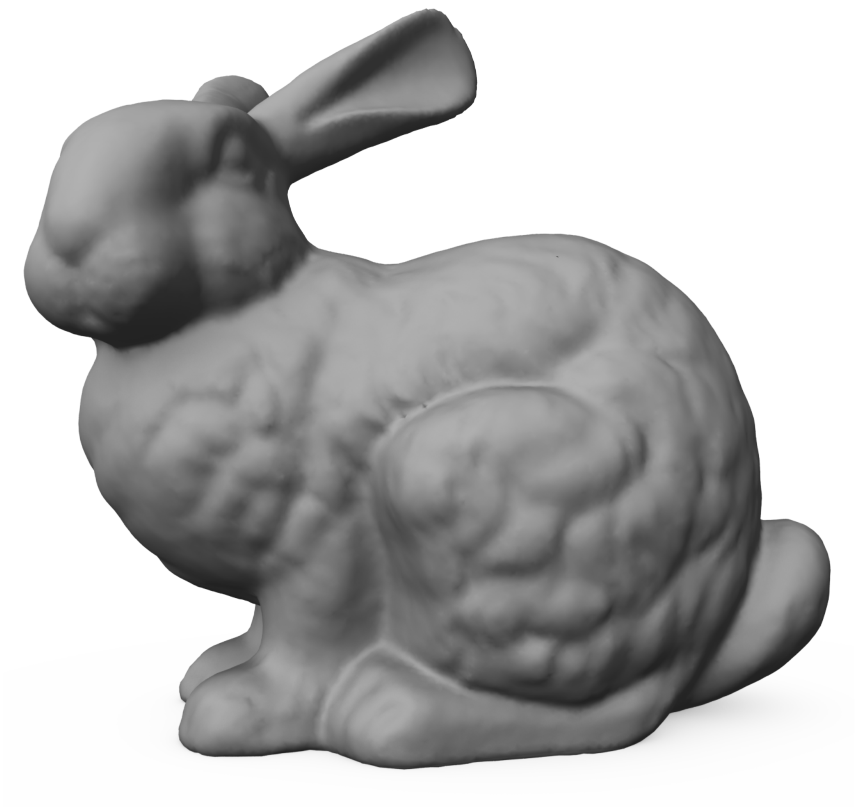
Sgng constructs a mesh that visually represents the input data more accurately than spsr. The latter tends to oversmooth the data (Fig. 4.20(a)) while sgng reconstructs finer details with the same number of vertices (Fig. 4.20(b)): The bulges on the side and the rear foot, and the bumps near the mouth of Stanford Bunny are more prominent for sgng than for spsr. The quality of the triangles that are constructed by sgng on the one hand and by spsr on the other hand is different (Fig. 4.21): Sgng constructs a fairly regular triangle mesh, that can be seen in a close-up of the left eye of Stanford Bunny (Fig. 4.21(b)). In contrast, spsr constructs many right angled triangles. Quite a few are even close to degenerate (Fig. 4.21(a)). However, this is caused by *marching cubes* that is known to construct a considerable amount of irregular triangles.

Holes that are present in the input data are reconstructed much better by sgng than by spsr: On the bottom of the original Stanford Bunny clay figure there are two holes for balancing the pressure during baking. Additionally, some parts of the bottom surface were not captured while scanning and are thus missing from the input points. Sgng correctly reconstructs the two holes (Fig. 4.32(b)) and reveals the missing data without any modifications to the learning parameters. In contrast, spsr bridges those parts and the accompanying trimming tool is needed to reopen at least the holes partially (Fig. 4.32(a)). Thus, regions with missing input data cannot be determined from an spsr reconstruction with sufficient certainty, whereas those regions are clearly apparent in an sgng reconstruction.

The difference in visual accuracy of sgng and spsr becomes also apparent in the reconstructions of Happy Buddha. The sgng reconstruction (Fig. 4.23) reproduces the wrinkles of the cloth better than the spsr reconstruction (Fig. 4.22). Spsr bridges the coat and the base on the right side of the statue. Furthermore, even the trimmed spsr reconstruction does not track the boundaries of the original object well, and some overshooting of the underlying distance function's zero-set becomes apparent: The right foot of Happy Buddha seems to be sunken into the base of the figure with spsr (Fig. 4.22) making it appear soft. Although there is a considerable amount of noise contained in the Buddha data set, with its current settings sgng fits a fairly smooth surface to the input points without creating spikes or other artifacts that are due to overfitting (Fig. 4.23). Furthermore,

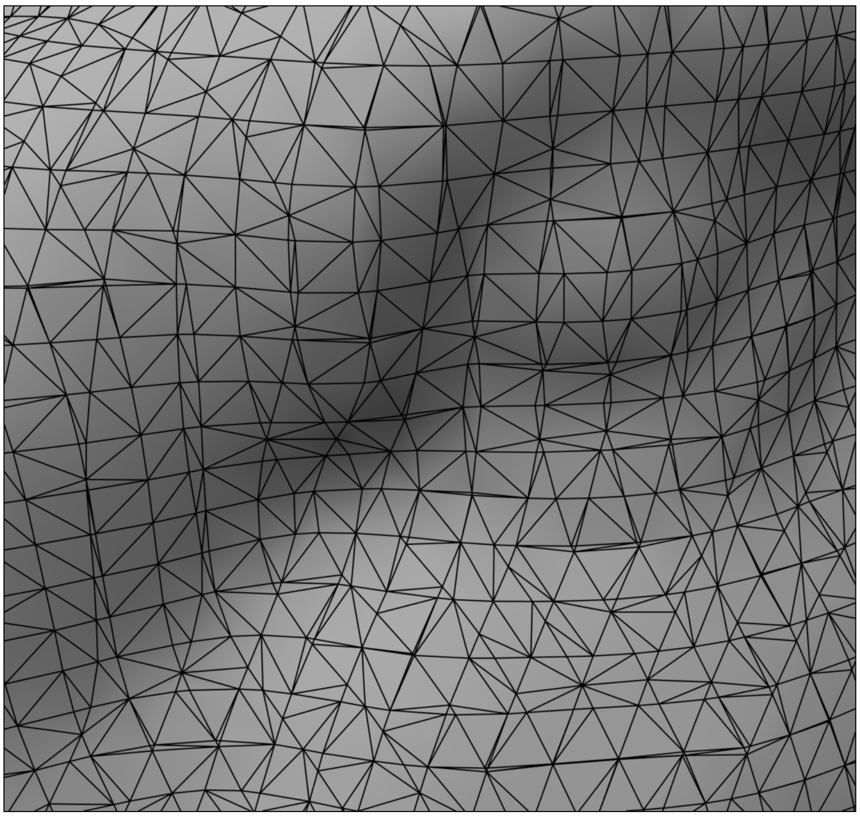


(a) Screened Poisson Surface Reconstruction.



(b) Surface-Reconstructing Growing Neural Gas.

Figure 4.20: Reconstructions of Stanford Bunny.


(a) Screened Poisson Surface Reconstruction.

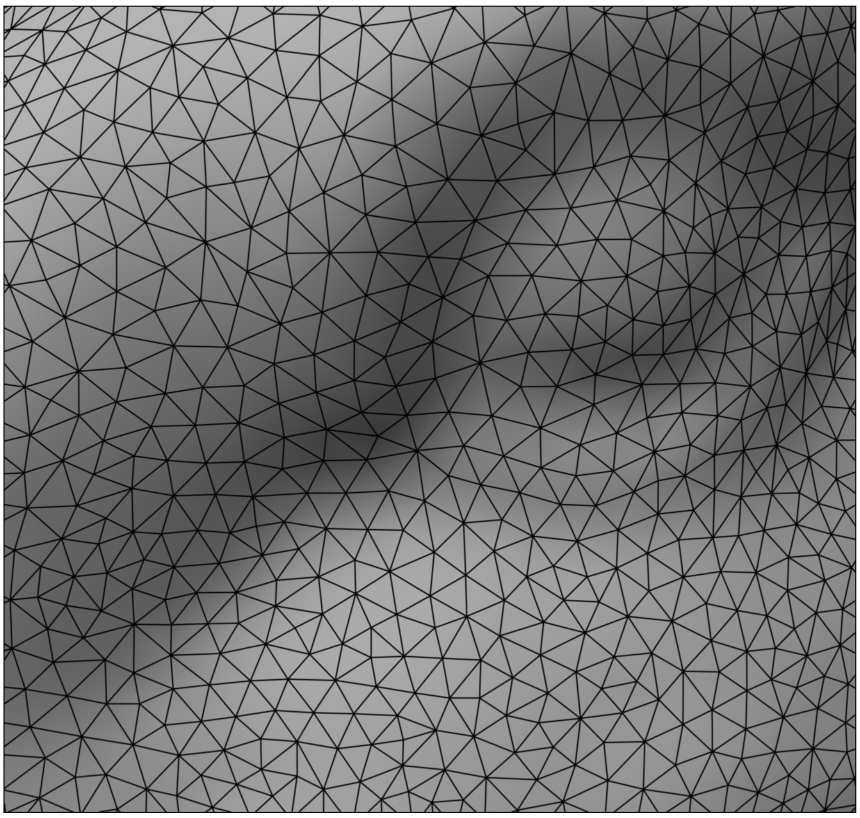

(b) Surface-Reconstructing Growing Neural Gas.

Figure 4.21: Close-up of the left eye of Stanford Bunny.

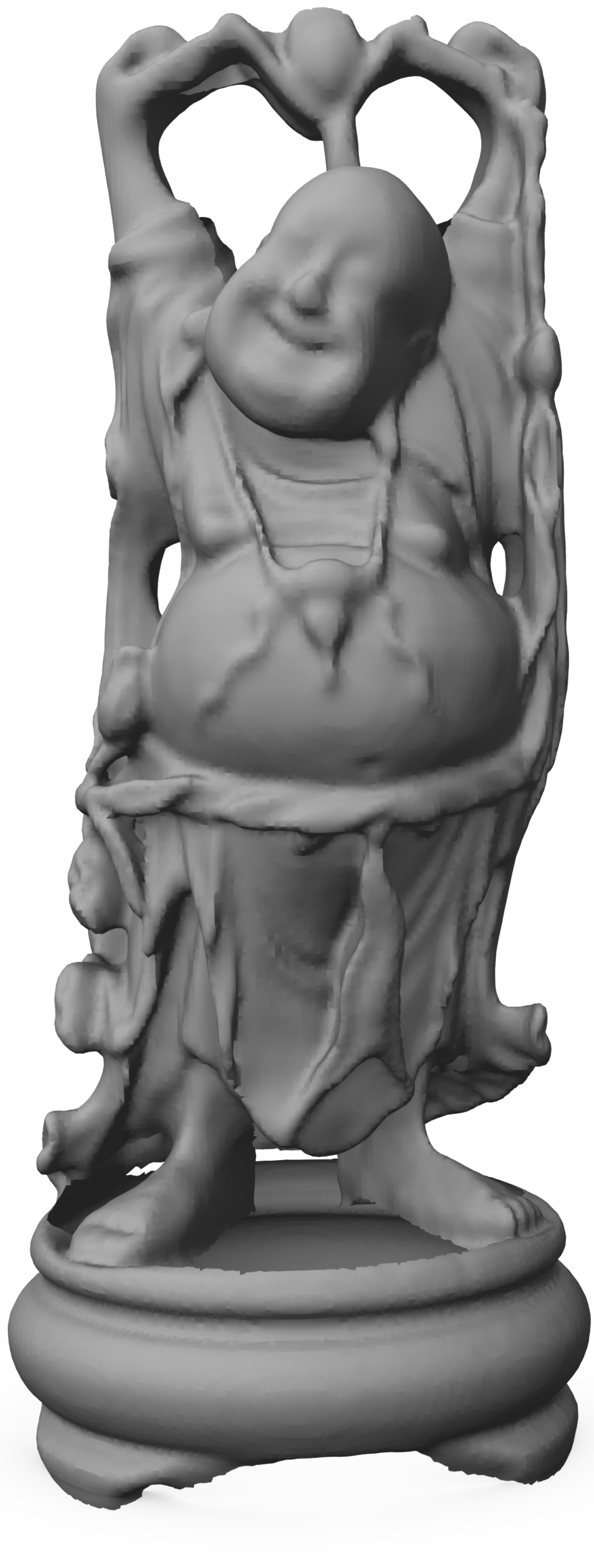


Figure 4.22: Spsr reconstruction of Happy Buddha.

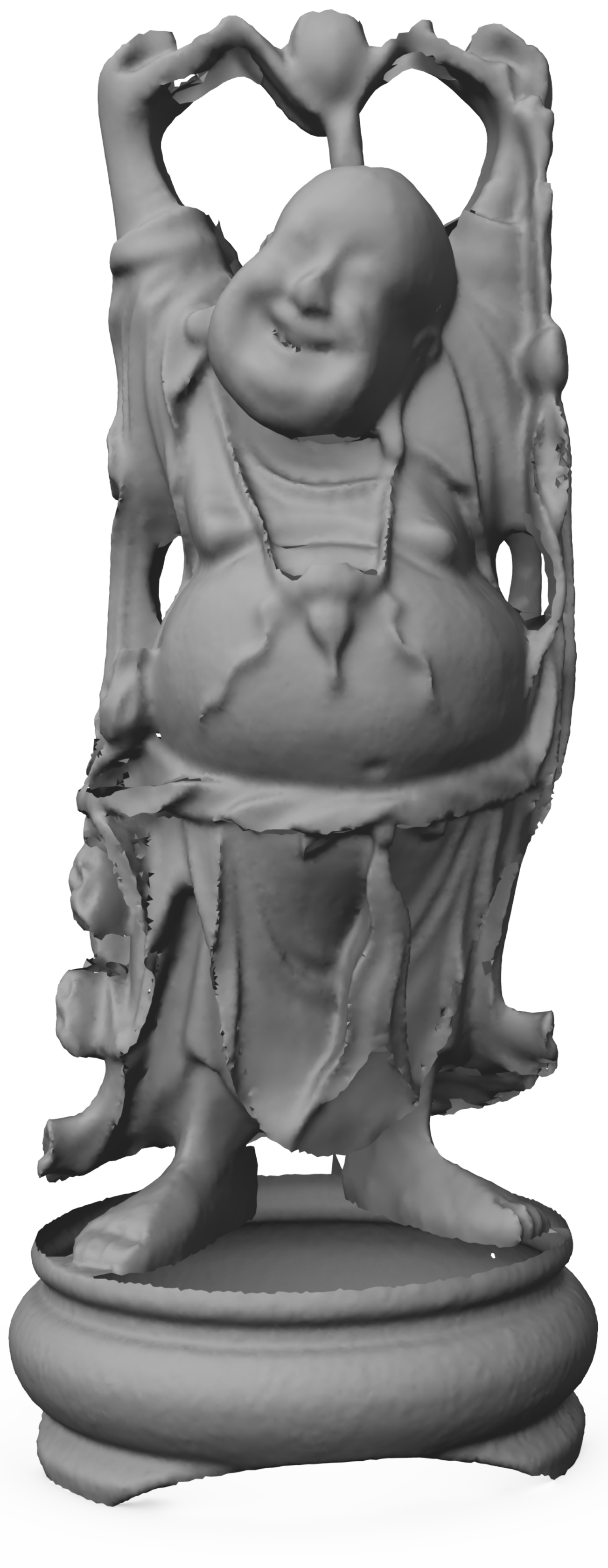
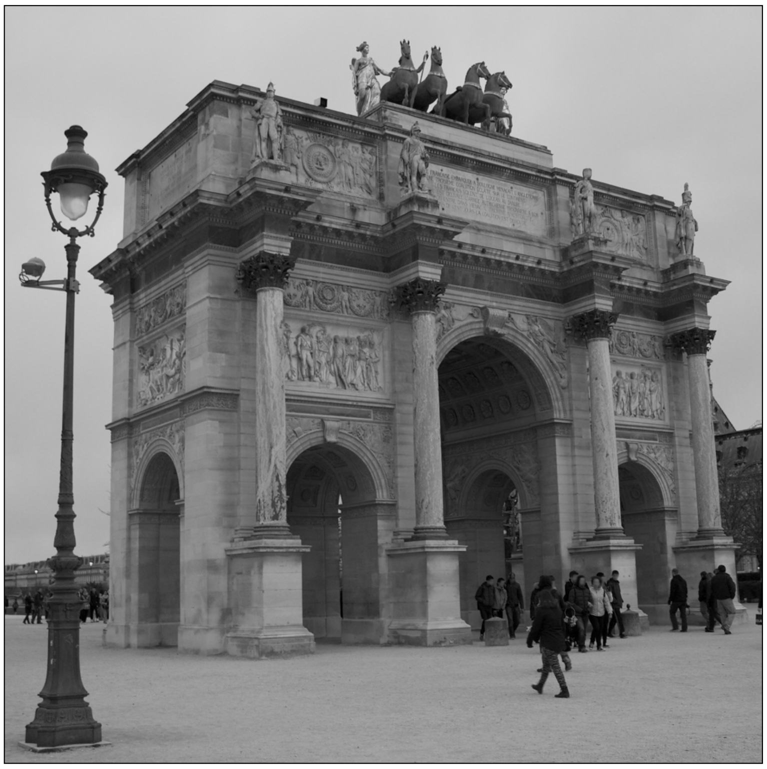
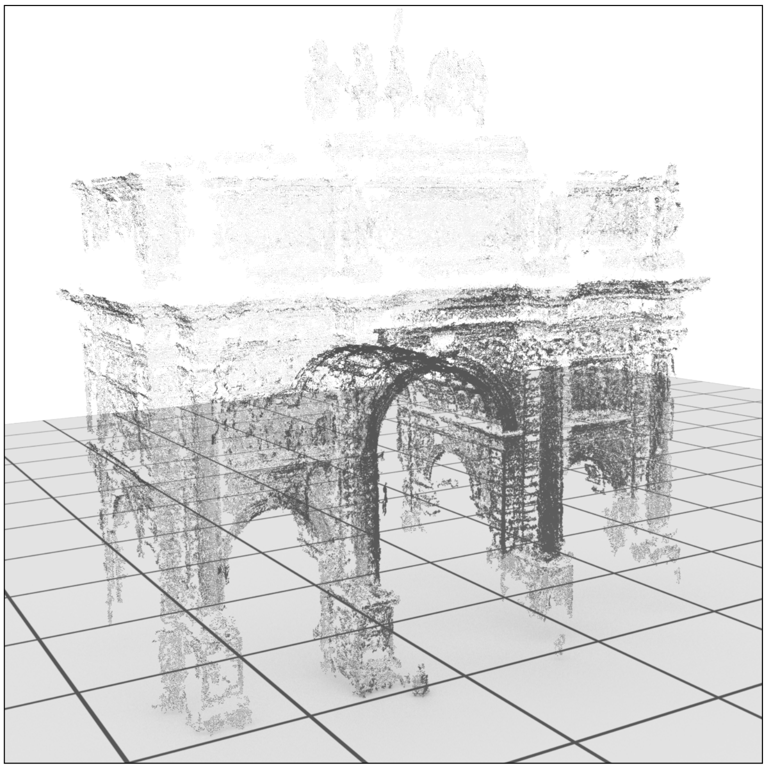


Figure 4.23: Sgng reconstruction of Happy Buddha.

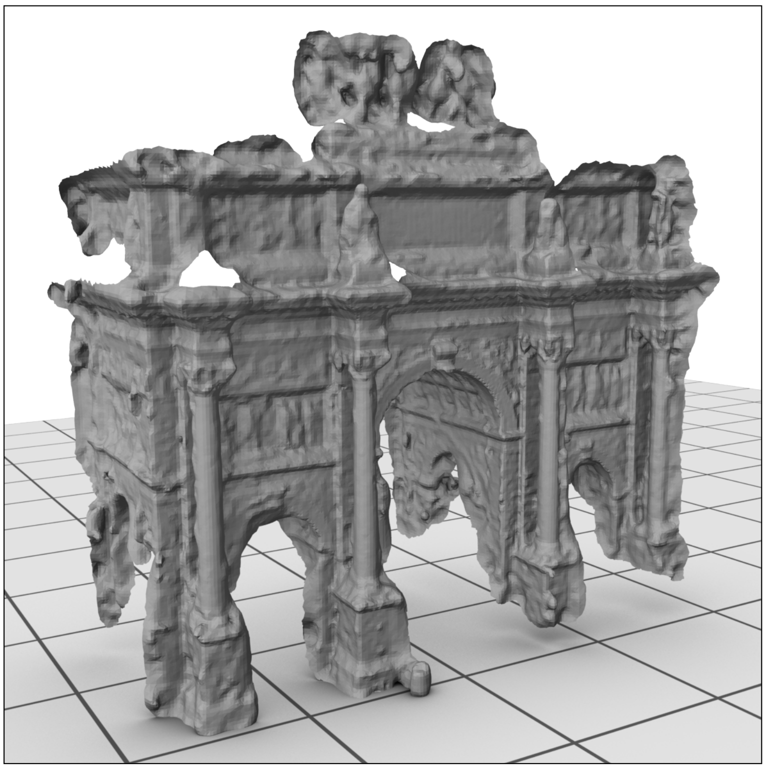


(a) Photo.



(b) Point Cloud.

Figure 4.24: Arch: Exemplary photo and complete point cloud.


(a) Screened Poisson Surface Reconstruction.

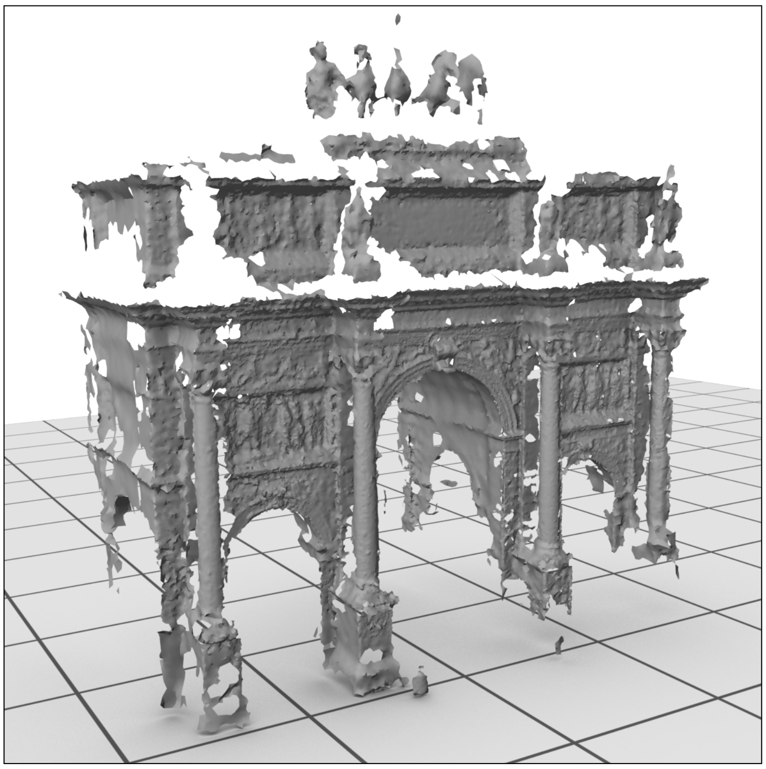
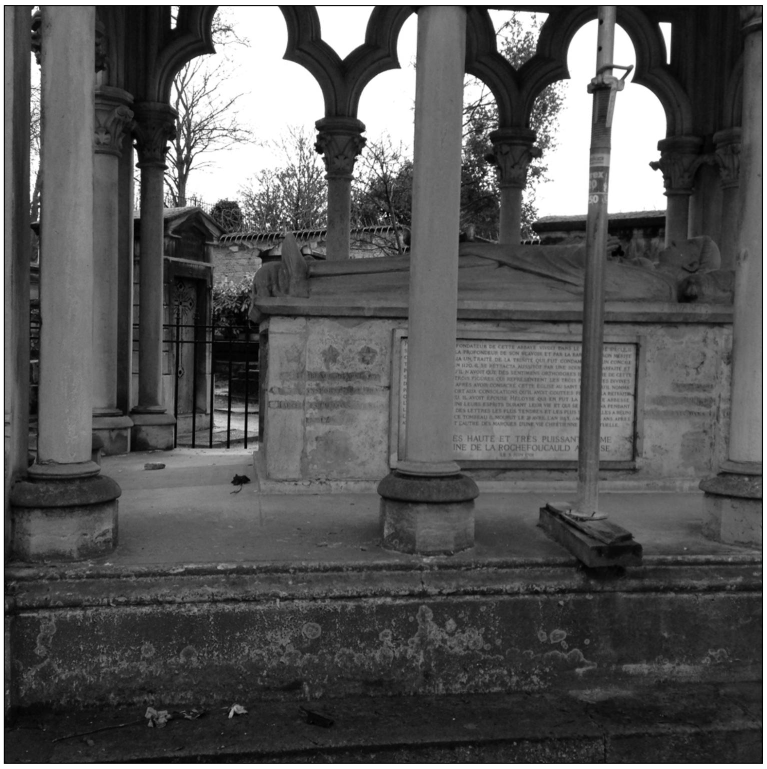
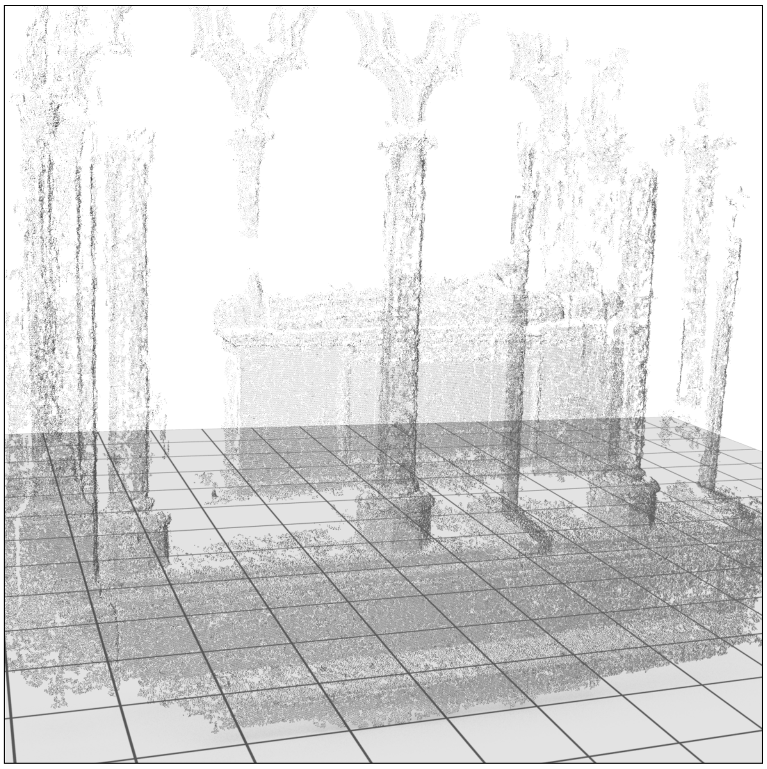

(b) Surface-Reconstructing Growing Neural Gas.

Figure 4.25: Arch: Constructed triangle meshes.

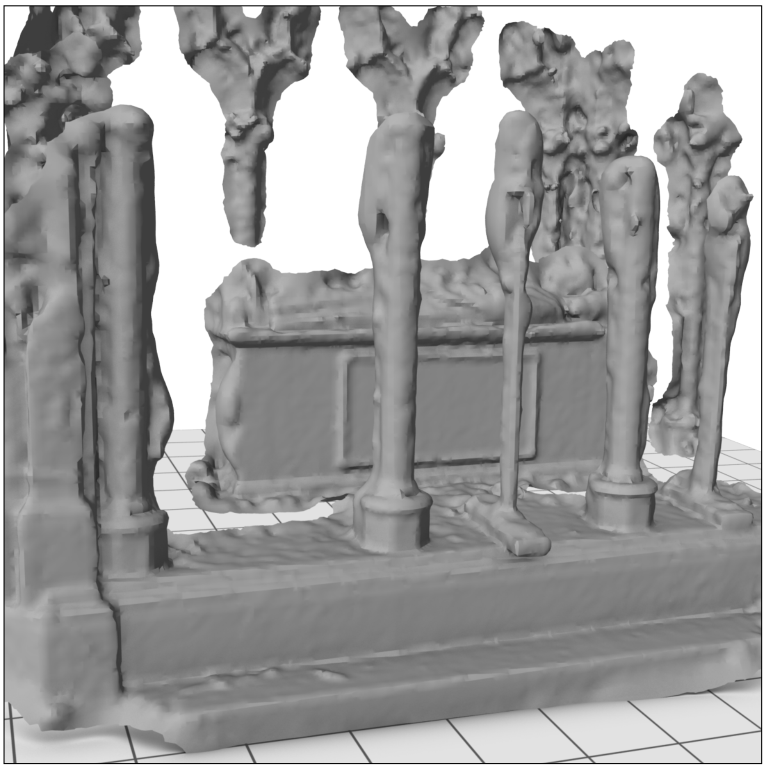


(a) Photo.



(b) Point Cloud.

Figure 4.26: Cemetery: Exemplary photo and complete point cloud.


(a) Screened Poisson Surface Reconstruction.

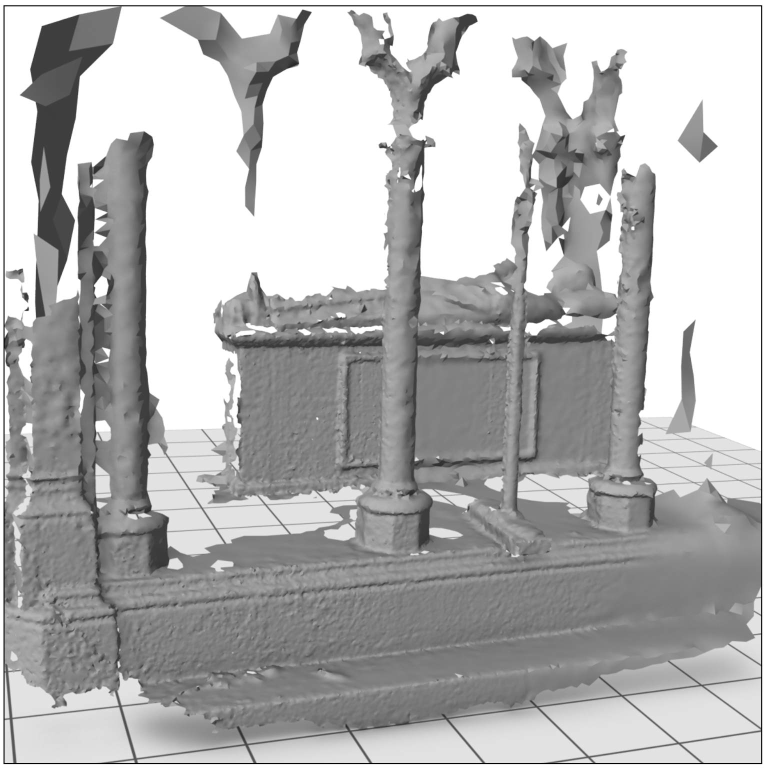
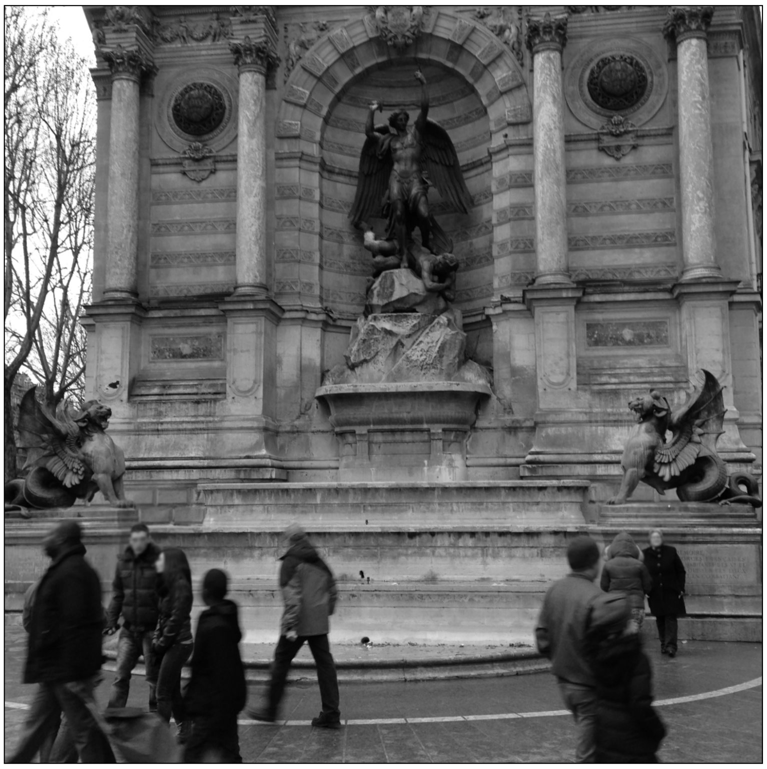

(b) Surface-Reconstructing Growing Neural Gas.

Figure 4.27: Cemetery: Constructed triangle meshes.


(a) Photo.

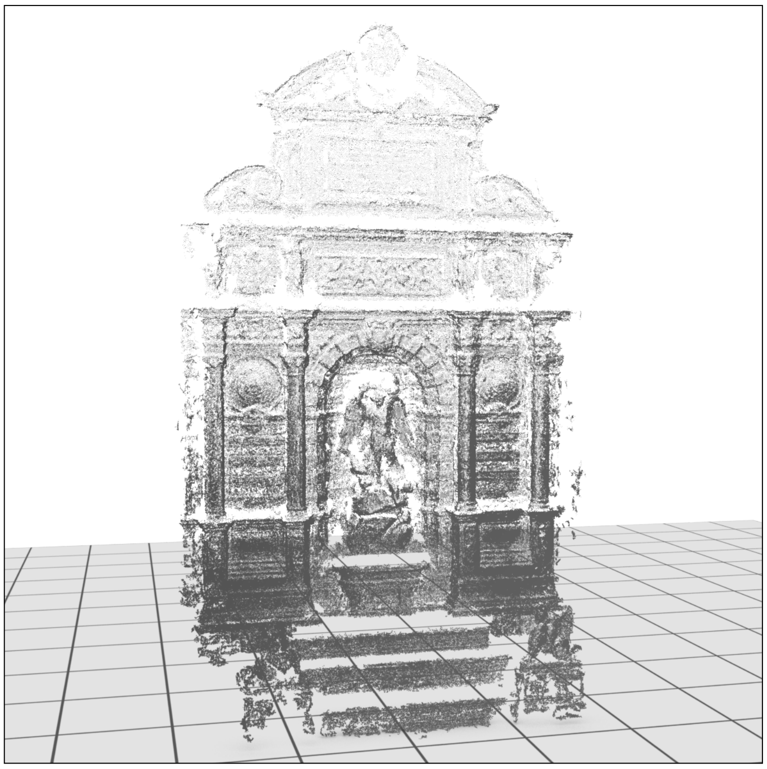
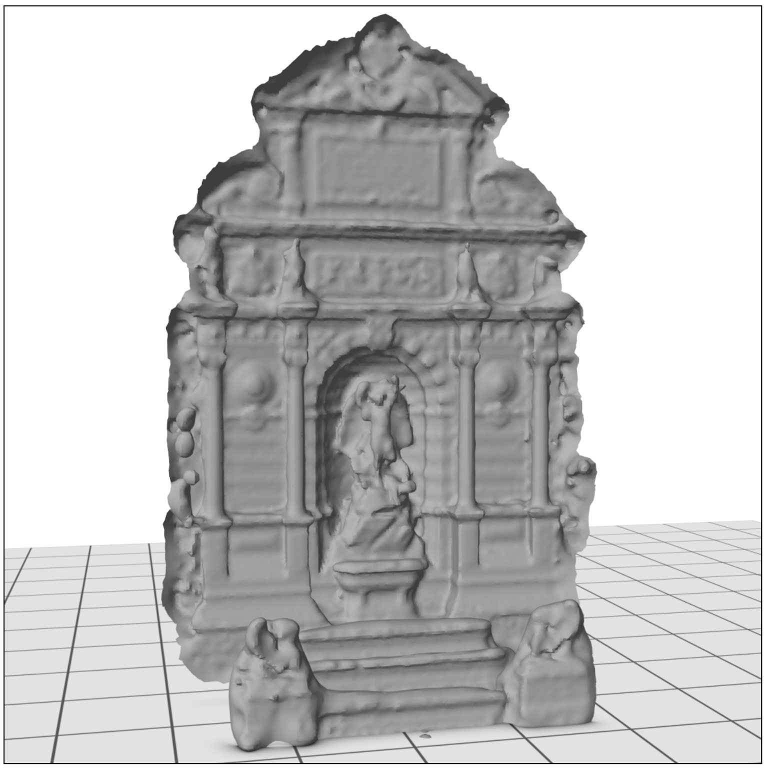
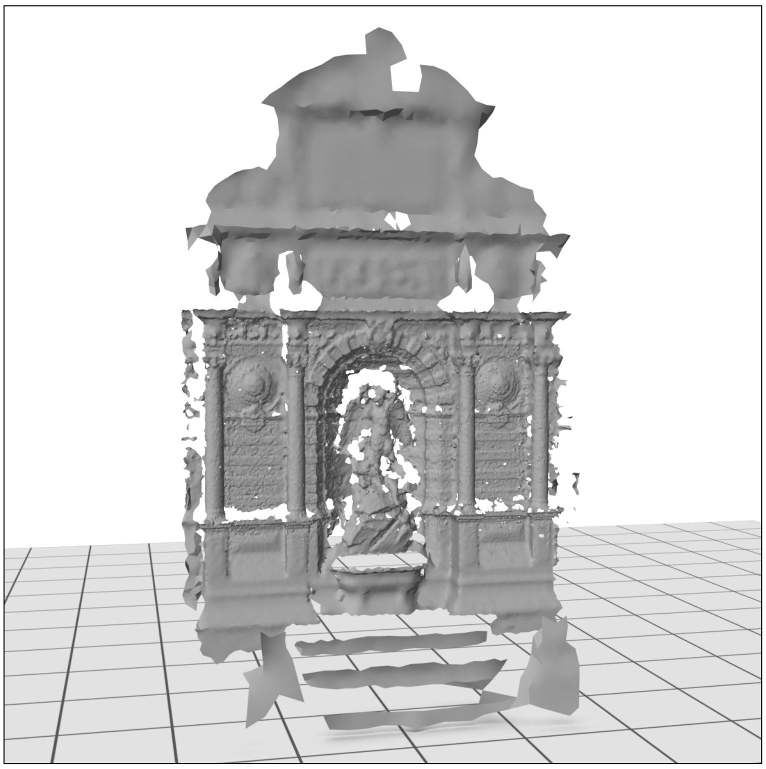

(b) Point Cloud.

Figure 4.28: Fountain: Exemplary photo and complete point cloud.

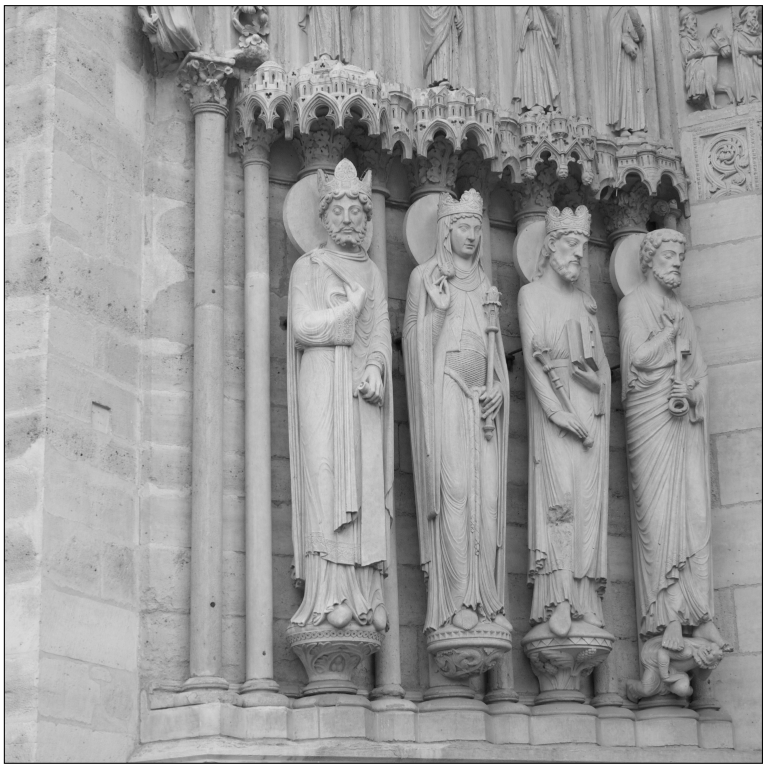


(a) Screened Poisson Surface Reconstruction.



(b) Surface-Reconstructing Growing Neural Gas.

Figure 4.29: Fountain: Constructed triangle meshes.


(a) Photo.

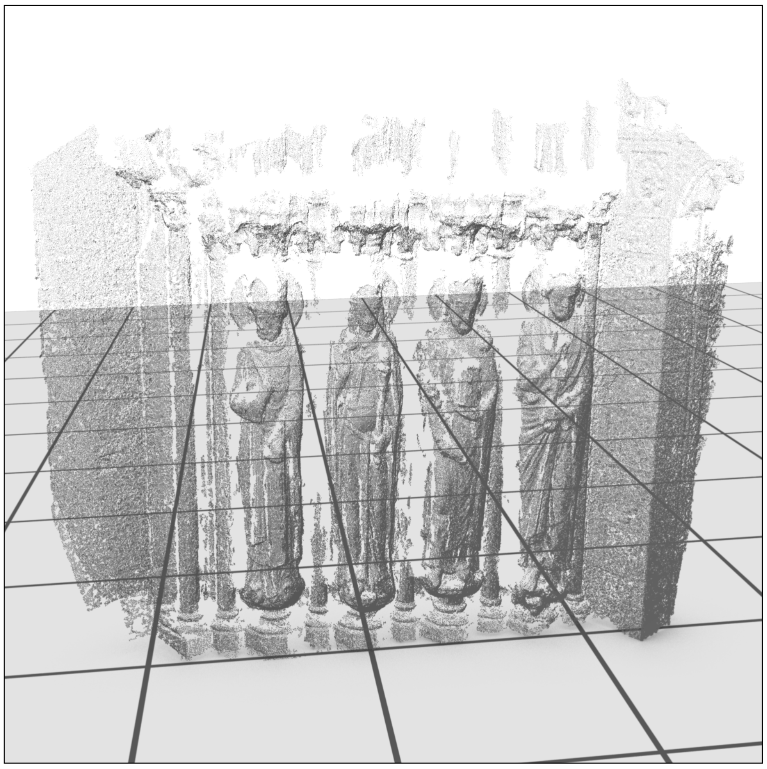
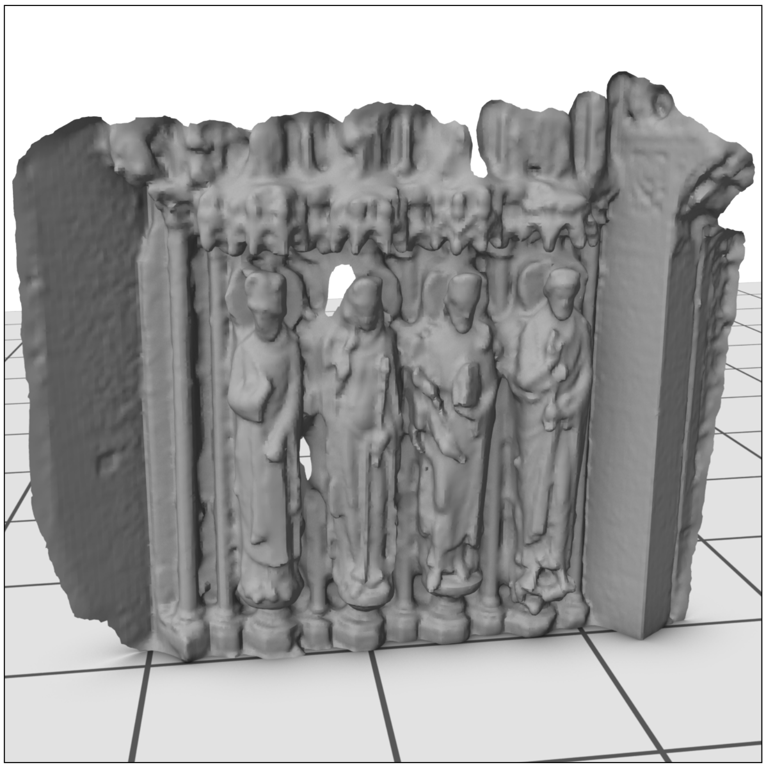

(b) Point Cloud.

Figure 4.30: Statues: Exemplary photo and complete point cloud.


(a) Screened Poisson Surface Reconstruction.

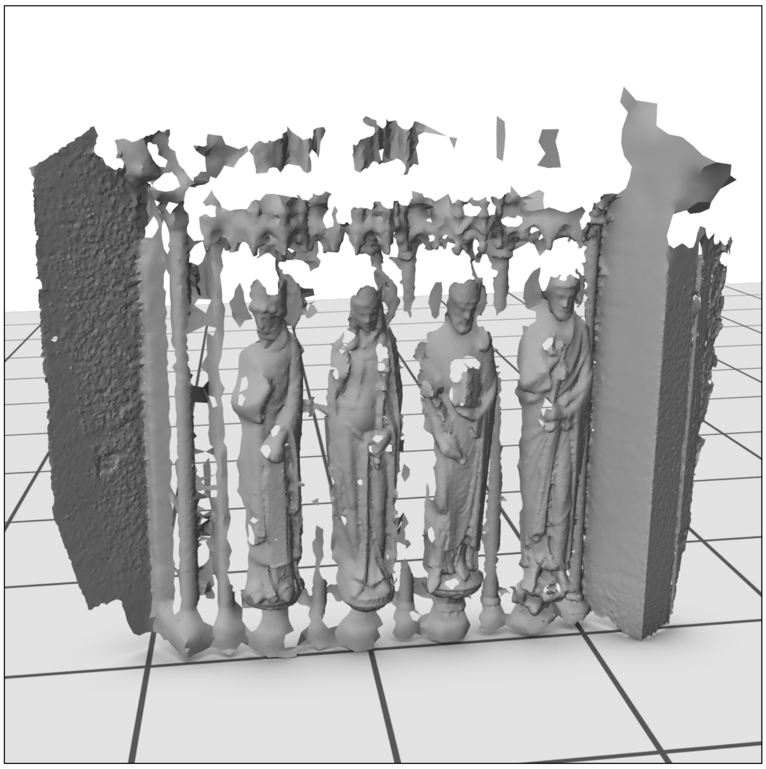

(b) Surface-Reconstructing Growing Neural Gas.

Figure 4.31: Statues: Constructed triangle meshes.

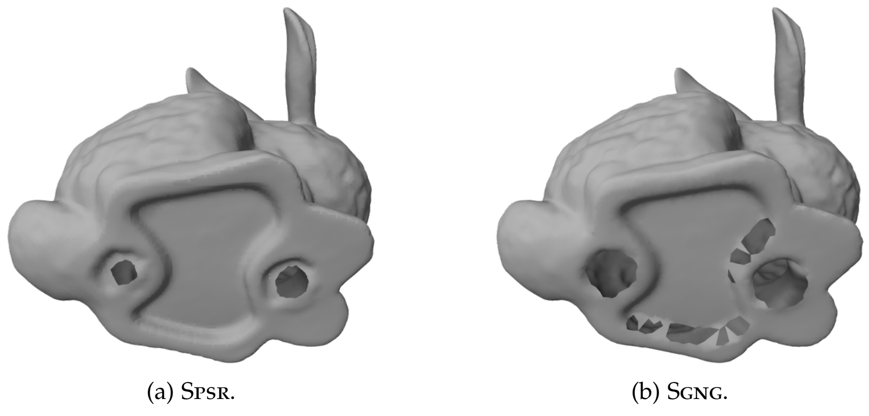

(a) Spsr. (b) Sgng.

Figure 4.32: View of the bottom of Stanford Bunny.

sgng tracks the boundary tightly within the variation of the input data's noise. Thus the reconstruction does not mislead a viewer about the materials of the original object. Finally, sgng does not bridge empty regions of the input data, except for one triangle in the back, erroneously bridging the coat and the base.

The reconstructions of the remaining, larger real-world objects further underline the visual accuracy of sgng reconstructions in comparison to spsr reconstructions. Parts of the original object that are not captured by the respective point cloud (Fig. 4.24(b), 4.26(b), 4.28(b), 4.30(b)) are bridged in the spsr reconstructions (Fig. 4.25(a), 4.27(a), 4.29(a), 4.31(a)), although the accompanying trimming tool was used with a high trimming value. Therefore, details in bridged regions might be missed since an operator cannot decide by examining the reconstruction where data has to be added. In contrast, the sgng reconstructions (Fig. 4.25(b), 4.27(b), 4.29(b), 4.31(b)) track the input point clouds well: Parts of the original object that are not captured by the respective point cloud are left empty in the constructed mesh. From these reconstructions an operator can decide where further data needs to be acquired in order to complete the reconstruction. Thus, eventually no details will be missing. Furthermore, the spsr reconstructions look smoothed. Due to this smoothness they appear to be rather small and made from soft materials. In combination with the already mentioned overshooting the spsr reconstructions look rather like a mold for the object than like the object itself. In contrast, the sgng reconstructions look crisp and provide a much better impression of material and size.

Triangle quality

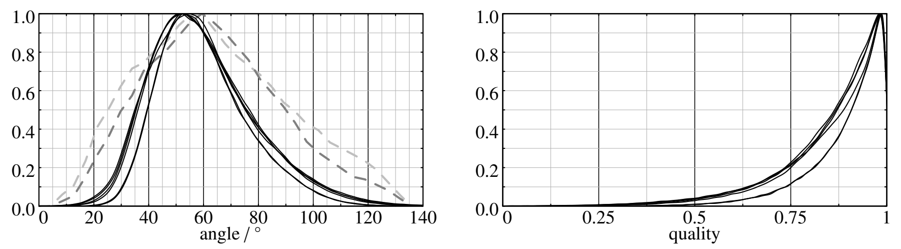
Sgng constructs fairly regular triangles: The normalized angle distributions of the individual experiments (Fig. 4.33(a) left, solid black) concentrate around 53°. This high regularity is reflected in a high triangle quality: The normalized quality distributions of the individual sgng experiments (Fig. 4.33(a) right) concentrate at $q \approx 0.98$. In contrast to these results, spsr features half-square triangles: The normalized angle distributions of the individual experiments (Fig. 4.33(b) left) concentrate around 45° and 89°. This is also reflected by the triangle quality: The normalized quality distributions of the individual SPSR experiments (Fig. 4.33(b) right) concentrate at $q \approx 0.84$.

The different triangle configurations that SGNG and SPSR create can be evaluated in greater detail using the contour plots in Fig. 4.34, 4.35. The plots give an overview of the frequency of triangle configurations, by identifying the most common 25 % (black), 50 % (gray), and 75 % (light gray) configurations of the triangles in the constructed mesh. It can clearly be seen that SGNG and SPSR produce significantly different distributions of triangle configurations. The most common 25 % of triangle configurations that are constructed by SGNG are very close to that of regular triangles for all data sets (Fig. 4.34(a), 4.34(c), 4.35(a), 4.35(c), 4.35(e), 4.35(g)). Even the most common 75 % of triangle configurations that are constructed by SGNG are still pretty close to that. In contrast, the most common 25 % of triangle configurations that are constructed by SPSR are very close to that of right-angled, isosceles triangles for all reconstructions (Fig. 4.34(b) 4.34(d), 4.35(b), 4.35(d), 4.35(f), 4.35(h)). Although the close to regular configurations are among the most common 50 % of triangle configurations for some SPSR reconstructions, there are also elongated, right-angled configurations in every SPSR reconstruction. Furthermore, even degenerate, right-angled configurations are among the most common 75 % of triangle configurations that are constructed by SPSR.

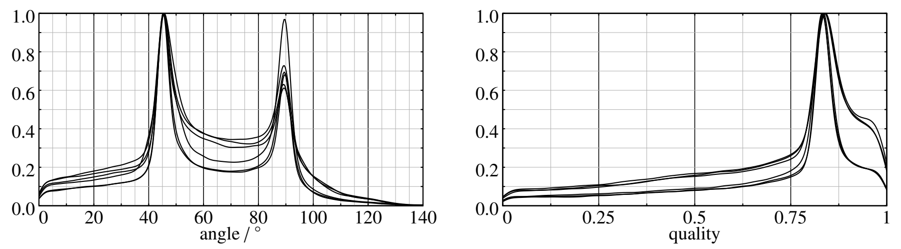
The irregular triangles constructed by SPSR are caused by *marching cubes* that is used to triangulate the zero-set of the computed distance function. In order to provide a less biased comparison, the angle distributions of SGNG are additionally compared to the results of the dictionary-learning approach by Xiong et al. [133] (Fig. 4.33(a) left, dashed gray) and to results of *Singular Cocone reconstruction* by Dey and Wang [38] (Fig. 4.33(a) left, dashed light gray). The curves are reproduced from the histograms published by Xiong et al. [133] in order to provide a direct comparison. The peaks in the angle distributions created by SGNG are located at a slightly smaller angle than in both the dictionary-learning approach and in *Singular Cocone reconstruction*. However, the distribution achievable with SGNG is much steeper, thus considerably fewer acute or obtuse triangles are created.

4.7 DISCUSSION

In this chapter SGNG has been proposed and evaluated, an online learning-based artificial neural network that iteratively constructs a triangle mesh from an unorganized point cloud representing an object's surface. SGNG instantly provides a rough approximation of the original surface that gets continuously refined up to an accurate reconstruction. Since all desired triangles are created during learning and not during

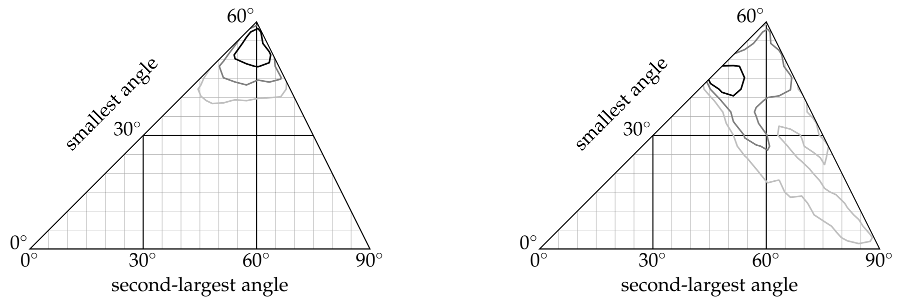


(a) Surface-Reconstructing Growing Neural Gas (solid black), dictionary learning [133] (dashed gray), Singular Cocone Dey and Wang [38] (dashed light gray). The latter two are reproduced from the histograms published by Xiong et al. [133].



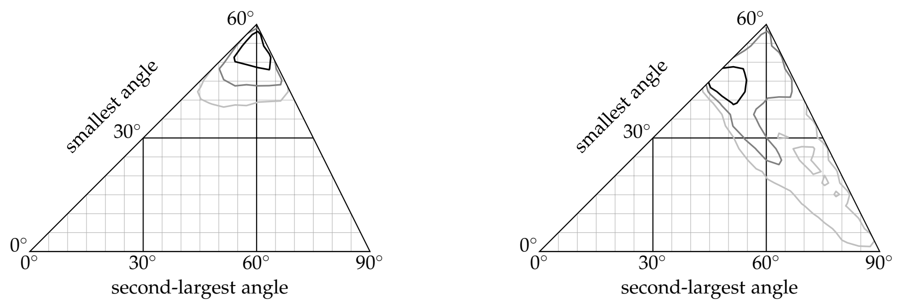
(b) Screened Poisson Surface Reconstruction.

Figure 4.33: Normalized angle and quality distributions.



(a) SGNG: Stanford Bunny.

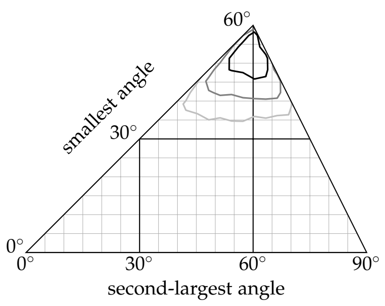
(b) SPSR: Stanford Bunny.



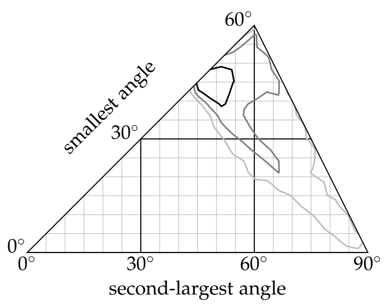
(c) SGNG: Happy Buddha.

(d) SPSR: Happy Buddha.

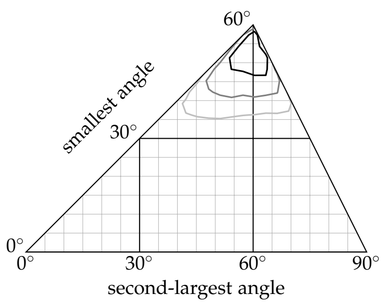
Figure 4.34: Contour plots displaying the relative frequency of triangle configurations that are created by SGNG and SPSR. The most common 25 % of configurations are enclosed by a black contour. The most common 50 % and 75 % of configurations are enclosed by a gray and a light gray contour, respectively.



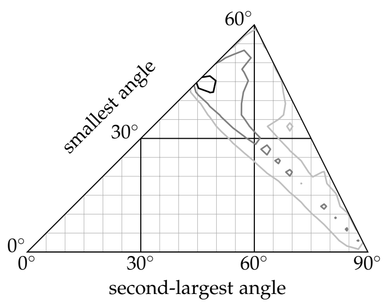
(a) Sgng: Arch.



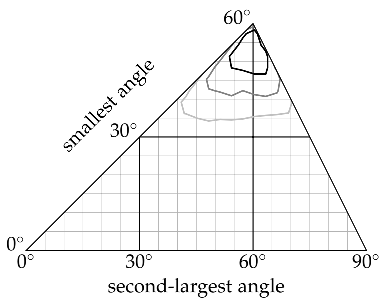
(b) spsr: Arch.



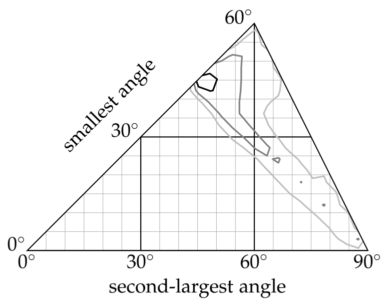
(c) Sgng: Cemetery.



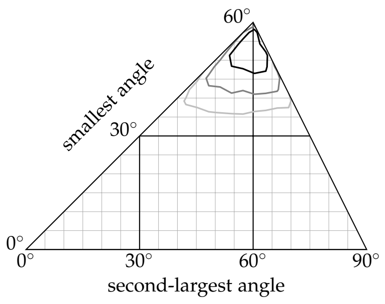
(d) spsr: Cemetery.



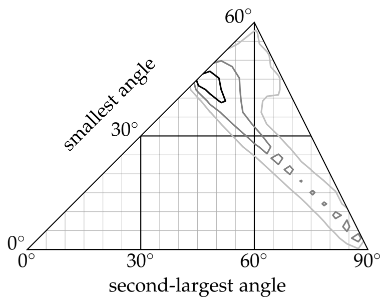
(e) Sgng: Fountain.



(f) spsr: Fountain.



(g) Sgng: Statues.



(h) spsr: Statues.

Figure 4.35: Contour plots displaying the relative frequency of triangle configurations that are created by sgng and spsr. The most common 25 % of configurations are enclosed by a black contour. The most common 50 % and 75 % of configurations are enclosed by a gray and a light gray contour, respectively.

post-processing, SGNG is well suited for an iterative pipeline where scanning, reconstruction, and visualization are executed in parallel: At any time modifications to the input data are incorporated instantly into the reconstruction without additional overhead. That way, with SGNG an operator or an automatic process will be able to direct further data acquisition without waiting for a reconstruction to be finished. In order to provide visually meaningful initial approximations SGNG tolerates self-intersections and that the constructed mesh may become non-manifold. However, adding a guarantee to SGNG that the constructed mesh will be manifold, once the input data is dense enough, is left for future work.

An extensive evaluation has confirmed that SGNG improves significantly upon its predecessor GSRM. Furthermore, results of a benchmark test have confirmed that SGNG can compete with other state-of-the-art surface reconstruction algorithms. Although SGNG is intended to be used as an online algorithm providing iterative preview while acquiring data, the reconstruction error of SGNG is low while the quality of the constructed mesh is high. SGNG visually represents the input data much better than SPSR, a state-of-the-art offline algorithm. The constructed meshes contain mostly regular triangles, outperforming SPSR, *Singular Cocone*, and a recent dictionary-learning approach. However, a comparison to several more recent surface reconstruction techniques is left for future work. SGNG is robust to noise as long as the variation of the noise is smaller than the average triangle size. Furthermore, the additional fitting step lets SGNG fit data boundaries tightly. Finally, the new topology learning enables SGNG to correctly represent empty regions of the input data.

SGNG does not extrapolate missing data: Holes are created whenever the vertex density exceeds the point density. Nevertheless, to overcome this, further input data can be added, and the learning parameters can be adjusted at any time during reconstruction. However, experiments indicate that the learning parameters of SGNG do not need to be specifically adjusted for each input data set: Leaving them unchanged for all experiments consistently produced good results.

Since SGNG uses a variant of CHL and approximates the positions of the input points, it cannot yet reconstruct very fine structures or sharp features, if these are sampled too sparsely: The latter will be smoothed, the former will lead to holes and wrinkles. With the current update rules, input data has to be very dense in such regions in order to avoid these errors. Approximating such features more accurately is left for future work.

It is very likely that the current implementation of SGNG leaves room for optimization: Profiling revealed that the 2-nearest neighbor search in the adaptive octree to find the best and the second-best matching vertices dominates running time. Furthermore, a considerable number of cache misses was reported. It is planned to examine the SGNG algorithm in order to find a way to address both bottlenecks and to further streamline the reconstruction process. Since SGNG operates only locally and evaluates only one input point at a time in a stochastic way, it is very likely that the algorithm can be executed massively in parallel, implemented as an out-of-core algorithm. Thus, even a GPU implementation is conceivable. In order to improve performance in terms of smaller reconstruction error and better reconstruction quality even further, other learning objectives or even dual representations of SGNG have to be examined.

It might be advantageous if the iteratively refined triangle meshes constructed by SGNG were fed back into the point extraction process. That way at first only a very coarse set of feature points has to be extracted, matched and converted into 3D points. The iteratively refined surface that is constructed by SGNG could then be used to direct extraction of a denser set of feature points. Thus, a combination of point extraction and SGNG could operate in an integrated, incremental coarse to fine approach, with reduced processing power and memory requirements—maybe even on mobile platforms.

5 LEARNING THE TEXTURE ASSIGNMENT*

If the input points that have been acquired for surface reconstruction provide color information about the original object, then the previously proposed *surface-reconstructing growing neural gas* (SGNG) can learn vertex colors from this information. Furthermore, if images of the original object are available that are registered to the input points, then these images can be used as textures for the constructed mesh.

A coloring and a texturing extension to SGNG are presented in this chapter. While the former is a straightforward way to add visual detail to the constructed mesh, the latter enables SGNG to automatically learn how to assign suitable high-resolution textures to the constructed triangles. By learning visibility from the input data instead of deriving it from the constructed mesh, SGNG reduces the number of noticeable occlusion artifacts to a minimum. Whenever new images become available, they are immediately incorporated into the model being learned, refining the result, without interrupting the reconstruction process. That way, even a textured mesh is available for visualization at any time during reconstruction.

5.1 PROBLEM STATEMENT

SGNG is originally intended to be used as a surface reconstruction algorithm using 3D input data that has been extracted from a set of images. The images can stem, for instance, from aerial photography using remotely-piloted aircraft, or from crowd work using digital cameras or smart phones. If the input images are mutually overlapping,

* Substantial parts of this chapter appeared in [120] Tom Vierjahn and Klaus Hinrichs. Surface-reconstructing growing neural gas: A method for online construction of textured triangle meshes. *Computers & Graphics*, pp. –, 2015. doi: 10.1016/j.cag.2015.05.016.
A first prototype was sketched in [123] Tom Vierjahn, Jan Roters, Manuel Moser, Klaus Hinrichs, and Sina Mostafawy. Online reconstruction of textured triangle meshes from aerial images. In *1st Eurographics Workshop Urban Data Modelling and Visualisation*, pp. 1–4, 2013. doi: 10.2312/UDMV/UDMV13/001-004.

then 3D input points can be extracted by stereophotogrammetry or by structure from motion [130] for subsequent surface reconstruction. For this purpose, at first, special feature points are identified in each image. The features are then matched across the images according to their feature descriptors so that each feature in a group of matching features represents the same point on the original object. Many suitable feature descriptors exist, including CenSurE [3], FREAK [4], BRIEF [24], SIFT [89], and DAISY [118]. After matching the features, epipolar geometry is used to compute the points' 3D positions and the cameras' extrinsic and intrinsic parameters from the matching features' 2D locations inside the images [107]. Real-world data is prone to imperfections like noise, drifts, or sensor inaccuracies. Feature matching may introduce additional artifacts like outliers. Thus, robust estimation techniques like FestGPU [106], a GPU implementation of RANSAC, [46] are used to compute as accurate a solution as possible efficiently.

The process sketched above creates a set of 3D input points for subsequent surface reconstruction and relates every point to a 2D location in two or more images. Thus, SGNG is not restricted to reconstruct an original object's shape and topology. SGNG can also learn the object's surface color. Related results have been presented for instance for reconstructions of historic sites in Rome from community photo collections [2, 48]. However, these approaches used fundamentally different reconstruction and texturing techniques.

Available data Extracting the set $\mathcal{P}$ of 3D points from a set $\mathcal{T}$ of images yields for each input image $T_i \in \mathcal{T}$ which points $\mathbf{p}_j \in \mathcal{P}$ are visible in it. Thus, the indicator function

$$[\mathbf{p}_j \text{ visible in } T_i] = \begin{cases} 1 & \text{, if } \mathbf{p}_j \text{ is visible in } T_i \\ 0 & \text{, otherwise} \end{cases} \tag{5.1}$$

is defined for all pairs of $T_i \in \mathcal{T}$ and $\mathbf{p}_j \in \mathcal{P}$. Consequently, a set $\mathcal{T}_{\mathbf{p}_j} \subseteq \mathcal{T}$ is defined for each input point $\mathbf{p}_j$ that contains only those images that show $\mathbf{p}_j$:

$$\mathcal{T}_{\mathbf{p}_j} = \{T_i \in \mathcal{T} \mid [\mathbf{p}_j \text{ visible in } T_i] = 1\} \quad .$$

Furthermore, point extraction computes a mapping

$$\mathbf{w}_\mathbf{p} : \mathcal{P} \times \mathcal{T} \mapsto \mathbb{R}^2 \quad \text{, with } \mathbf{w}_\mathbf{p}(\mathbf{p}_j, T_i) = \begin{cases} \mathbf{w}_{\mathbf{p}_j, T_i} & \text{, if } T_i \in \mathcal{T}_{\mathbf{p}_j} \\ \text{undefined} & \text{, otherwise} \end{cases} \quad ,$$

yielding a point's 2D location $\mathbf{w}_{\mathbf{p}_j, T_i}$ inside an image T_i if the point $\mathbf{p}_j$ is visible in the image. Determining the color at $\mathbf{w}_{\mathbf{p}_j, T_i}$ in the image T_i is straightforward. Let each image constitute a mapping from 2D locations inside the image to colors, for instance in RGB space,

$$T_i : \mathbb{R}^2 \mapsto \mathbb{R}^3 \quad \text{, with}$$

$$\mathrm{T}_i(\mathbf{w}_{\mathbf{p}_j,\mathrm{T}_i}) = \begin{cases} \text{color at } \mathbf{w}_{\mathbf{p}_j,\mathrm{T}_i} & \text{, if } \mathbf{w}_{\mathbf{p}_j,\mathrm{T}_i} \text{ inside image} \\ \text{undefined} & \text{, otherwise} \end{cases} \quad .$$

If $\mathbf{w}_{\mathbf{p}_j,\mathrm{T}_i}$ has sub-pixel accuracy, then a suitable interpolation scheme needs to be applied, e.g., linear interpolation. Finally, a color, for instance in RGB space, is assigned to each point by a mapping

$$\mathbf{c}_\mathbf{p} : \mathcal{P} \mapsto \mathbb{R}^3 \quad \text{, with } \mathbf{c}_\mathbf{p}(\mathbf{p}_j) = \frac{1}{|\mathcal{T}_{\mathbf{p}_j}|} \sum_{\mathrm{T}_i \in \mathcal{T}_{\mathbf{p}_j}} \mathrm{T}_i(\mathbf{w}_{\mathbf{p}_j,\mathrm{T}_i}) \quad ,$$

yielding the average color of the pixels that are showing a point. The shorthand $\mathbf{c}_{\mathbf{p}_j} = \mathbf{c}_\mathbf{p}(\mathbf{p}_j)$ is used in the remainder of this dissertation.

5.2 LEARNING VERTEX COLORS AND TEXTURE COORDINATES

SGNG approximates an original object's shape by learning vertex positions from the input points' positions by an algorithm that—intuitively—resembles a variant of k-means clustering [87, 90] using stochastic gradient descent. Learning vertex colors and interpolating them across the triangles as an approximation for the original object's color can be integrated into the reconstruction algorithm in a similar fashion. The same applies to learning texture coordinates inside the individual images. The derivation is outlined in the following.

Objective function

SGNG as a learning algorithm minimizes an objective function. To be precise, Erwin et al. [45] prove that there does not exist a single energy function for gradient descent in a *self-organizing map* (SOM). A similar argument might apply to SGNG, since it uses a neighbor learning scheme that is related to the one used in SOM. However, a closer examination lies beyond the scope of this dissertation. Nevertheless, for the sake of an intuitive illustration, existence of such a single energy function is assumed here. Let E denote this energy function

$$E : \mathcal{V}^* \mapsto \mathbb{R} \quad \text{, with } E(\mathcal{V}) = \sum_{\mathbf{v}\in\mathcal{V}} \sum_{\mathbf{p}\in\mathcal{P}_\mathbf{v}} \|\mathbf{v}-\mathbf{p}\|^2 \quad ,$$

where $\mathcal{V}^*$ denotes the set of all conceivable vertex sets, and where $\mathcal{P}_\mathbf{v} \subseteq \mathcal{P}$ denotes the set of all points that are located in the Voronoi cell of vertex $\mathbf{v}$. Applying gradient descent leads to the update rule

$$\mathbf{v} := \mathbf{v} - \beta' \cdot \nabla E = \mathbf{v} - \beta \cdot \sum_{\mathbf{p}\in\mathcal{P}_\mathbf{v}} (\mathbf{v}-\mathbf{p}) \qquad , \ \forall \mathbf{v} \in \mathcal{V} \quad .$$

Presenting the points one by one, one after the other, in random order, leads to a stochastic gradient descent, with the already known update rule for the best matching vertex $\mathbf{v}_\mathrm{b}$ according to the point $\mathbf{p}_{\xi_t}$ that is randomly selected in iteration t

$$\mathbf{v}_\mathrm{b} := \mathbf{v}_\mathrm{b} - \beta \cdot (\mathbf{v}_\mathrm{b} - \mathbf{p}_{\xi_t}) \quad .$$

A beautiful derivation of the above is presented by Singer and Warmuth [112], but for training a mixture of Gaussians.

Vertex colors With this intuition, learning vertex colors is derived, leading to an approach that is similar to the one presented by Orts-Escolano et al. [101, 102]. Let $\mathbf{c_v}$ denote the mapping to be learned, assigning a color, for instance in RGB space, to a vertex

$$\mathbf{c_v} : \mathcal{V} \mapsto \mathbb{R}^3 \quad ,$$

with $\mathbf{c_v}(\mathbf{v}_i) =$ undefined initially. The shorthand $\mathbf{c}_{\mathbf{v}_i} = \mathbf{c_v}(\mathbf{v}_i)$ is used in the remainder of this dissertation. Vertex colors are optimized independently of the vertex positions, in order to avoid that color influences the constructed shape. This leads to an energy function

$$E' : \mathcal{V}^* \mapsto \mathbb{R} \quad , \text{with } E'(\mathcal{V}) = \sum_{\mathbf{v} \in \mathcal{V}} \sum_{\mathbf{p} \in \mathcal{P}_\mathbf{v}} \|\mathbf{v} - \mathbf{p}\|^2 + \|\mathbf{c_v} - \mathbf{c_p}\|^2 \quad .$$

According to the above rationale an additional update rule for vertex colors is derived

$$\mathbf{c}_{\mathbf{v}_\mathrm{b}} := \mathbf{c}_{\mathbf{v}_\mathrm{b}} - \beta \cdot (\mathbf{c}_{\mathbf{v}_\mathrm{b}} - \mathbf{c}_{\mathbf{p}_{\xi_t}}) \quad .$$

Taking the neighbor learning of SGNG into account and replacing vertex positions with vertex colors leads to the color-learning rule for the directly connected neighbors of $\mathbf{v}_\mathrm{b}$

$$\mathbf{c}_{\mathbf{v}_n} := \mathbf{c}_{\mathbf{v}_n} - \eta \cdot (\mathbf{c}_{\mathbf{v}_n} - \mathbf{c}_{\mathbf{p}_{\xi_t}}) \quad , \forall \mathbf{v}_n \in \mathcal{N}_{\mathbf{v}_\mathrm{b}} \quad .$$

If $\mathbf{c}_{\mathbf{v}_i} =$ undefined, $\mathbf{c}_{\mathbf{p}_{\xi_t}}$ is assigned to $\mathbf{c}_{\mathbf{v}_i}$ as an initial approximation.

Texture coordinates The above derivation can be reused for letting the vertices in the constructed mesh learn texture coordinates from the 2D locations of the input points in the images. Then, SGNG keeps track of a mapping

$$\mathbf{w_v} : \mathcal{V} \times \mathcal{T} \mapsto \mathbb{R}^2$$

yielding a vertex' learned texture coordinates $\mathbf{w_v}(\mathbf{v}_j, \mathrm{T}_i)$ inside an image T_i, with $\mathbf{w_v}(\mathbf{v}_j, \mathrm{T}_i) =$ undefined initially. The shorthand $\mathbf{w}_{\mathbf{v}_j,\mathrm{T}_i} = \mathbf{w_v}(\mathbf{v}_j, \mathrm{T}_i)$ is used in the remainder of this dissertation.

Applying the same rationale that was used to derive the color learning rules leads to update rules for texture coordinates: For the best matching vertex $\mathbf{v}_\mathrm{b}$

$$\mathbf{w}_{\mathbf{v}_\mathrm{b},\mathrm{T}_i} := \mathbf{w}_{\mathbf{v}_\mathrm{b},\mathrm{T}_i} - \beta \cdot (\mathbf{w}_{\mathbf{v}_\mathrm{b},\mathrm{T}_i} - \mathbf{w}_{\mathbf{p}_{\xi_t},\mathrm{T}_i}) \quad , \forall \mathrm{T}_i \in \mathcal{T}_{\mathbf{p}_{\xi_t}} \quad ,$$

and for the directly connected neighbors of $\mathbf{v}_\mathrm{b}$

$$\mathbf{w}_{\mathbf{v}_n,\mathrm{T}_i} := \mathbf{w}_{\mathbf{v}_n,\mathrm{T}_i} - \eta \cdot (\mathbf{w}_{\mathbf{v}_n,\mathrm{T}_i} - \mathbf{w}_{\mathbf{p}_{\xi_t},\mathrm{T}_i}) \quad , \forall \mathrm{T}_i \in \mathcal{T}_{\mathbf{p}_{\xi_t}} \, , \forall \mathbf{v}_n \in \mathcal{N}_{\mathbf{v}_\mathrm{b}} \quad .$$

If $\mathbf{w}_{\mathbf{v}_j,\mathrm{T}_i} =$ undefined, $\mathbf{w}_{\mathbf{p}_{\xi_t},\mathrm{T}_i}$ is assigned to $\mathbf{w}_{\mathbf{v}_j,\mathrm{T}_i}$ as an approximation.

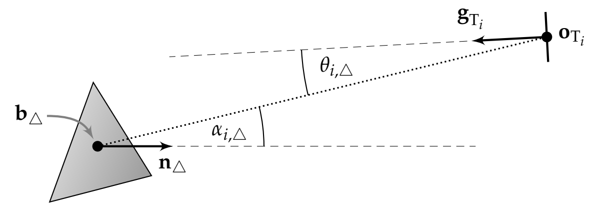


Figure 5.1: Selecting the image T_i that provides the most perpendicular view of the triangle $\triangle$.

Practical considerations

Learning vertex colors or texture coordinates as presented above and interpolating them across the triangles leads to a colored or textured triangle mesh—given that suitable textures for the triangles are selected. However, the level of detail in terms of color and the accuracy in terms of texture coordinates are coupled to the number and thus to the size of the constructed triangles, since vertex colors and texture coordinates are interpolated across them: The smaller the triangles, the more details in terms of color and texture coordinates are achievable. Thus, a large number of triangles is required for visually appealing, sharp, colored or textured results containing fine details, although a rather low number of triangles might be sufficient to represent a desired level of geometric detail. This coupling is also apparent in the results of other algorithms that determine vertex colors from the input points, for instance, in SSD-C [22]. Nevertheless, especially integrating color learning into SGNG significantly improves the visual quality of the results without increasing implementation complexity or adding a severe overhead to the reconstruction algorithm. However, in the next section a texturing approach is presented, that is well suited even for low-resolution triangle meshes.

5.3 assigning suitable textures

In order to provide better accuracy, SGNG computes texture coordinates for the vertices by projecting the constructed triangles into the input images using the previously determined attributes of the cameras. In order to avoid noticeable artifacts, the image that is assigned as a texture to a given triangle has to be carefully selected from the set of input images. Blending, and leveling techniques [17, 34, 56] should be applied to balance different exposure and lighting conditions. There are many such techniques readily available. However, integrating them is left for future work.

Finding the most perpendicular view

In order to minimize perspective distortions SGNG aims at assigning the image as a texture to a triangle that provides the most perpendicular view of the triangle. For this purpose, a suitable metric $\perp(\triangle, T_i)$ that is similar to the *form factor* in Radiosity [59] is derived from the position $\mathbf{o}_{T_i}$ and the unit gaze direction $\mathbf{g}_{T_i}$ of the original camera that captured $T_i \in \mathcal{T}$, and from the barycenter $\mathbf{b}_\triangle$ and the unit normal $\mathbf{n}_\triangle$

of the triangle $\triangle$ (Fig. 5.1):

$$\perp(\triangle, \mathrm{T}_i) = \frac{\big(\mathbf{n}_\triangle \cdot (\mathbf{o}_{\mathrm{T}_i} - \mathbf{b}_\triangle)\big) \cdot \big(\mathbf{g}_{\mathrm{T}_i} \cdot (\mathbf{b}_\triangle - \mathbf{o}_{\mathrm{T}_i})\big)}{(\mathbf{o}_{\mathrm{T}_i} - \mathbf{b}_\triangle) \cdot (\mathbf{o}_{\mathrm{T}_i} - \mathbf{b}_\triangle)}$$

$$= \cos \alpha_{i,\triangle} \cdot \cos \theta_{i,\triangle} \quad .$$

The image $\mathrm{T}_\triangle$ to be used for texturing a given triangle is then found by maximizing $\perp(\triangle, \mathrm{T}_i)$ over all $\mathrm{T}_i \in \mathcal{T}$

$$\mathrm{T}_\triangle = \arg\max_{\mathrm{T}_i \in \mathcal{T}} \perp(\triangle, \mathrm{T}_i) \quad .$$

While maximizing only $\perp(\triangle, \mathrm{T}_i)$ works well for reconstructions from clean scans of simple, convex objects, it is not well suited for the general case, since it does not handle occlusions, neither by other objects nor by self-occlusion. Therefore, real-world applications will suffer from noticeable artifacts. These artifacts can be avoided with existing techniques only if occluding surface parts are represented by the input points *and* if those are reconstructed accurately. Both can hardly be guaranteed in real-world applications. Since SGNG reconstructs objects incrementally and in a stochastic way, occluding triangles are not necessarily created accurately at any time. Fortunately, visibility information is implicitly encoded in the input points and can therefore be learned during SGNG reconstruction.

Avoiding occlusion artifacts

Each vertex learns its visibility in an image from the input points. The visibility is then used to select the image that is suitable for texturing a triangle without creating noticeable occlusion artifacts, even if occluding surfaces are not represented in the input points.

Let $\mathcal{P}_\mathbf{v}$ denote the set of all points that are located in the Voronoi cell of a vertex $\mathbf{v}$. Then, the probability that this vertex is visible in a given image T_i is estimated by the ratio of the points of $\mathcal{P}_\mathbf{v}$ that are visible in T_i

$$P(\mathbf{v} \text{ visible in } \mathrm{T}_i \mid \mathrm{T}_i) \approx \frac{\sum_{\mathbf{p} \in \mathcal{P}_\mathbf{v}} [\mathbf{p} \text{ visible in } \mathrm{T}_i]}{|\mathcal{P}_\mathbf{v}|} \quad ,$$

with $[\mathbf{p} \text{ visible in } \mathrm{T}_i]$ denoting the indicator function (Eq. 5.1). Having learned the visibility, optimization is extended to assign the image $\mathrm{T}_\triangle$ as a texture to a triangle $\triangle$ with vertices $\mathbf{v} \in \triangle$ that provides the most perpendicular view of the triangle *and* that most likely shows the unoccluded triangle:

$$\mathrm{T}_\triangle = \arg\max_{\mathrm{T}_i \in \mathcal{T}} \left(\perp(\triangle, \mathrm{T}_i) \cdot \prod_{\mathbf{v} \in \triangle} P(\mathbf{v} \text{ visible in } \mathrm{T}_i \mid \mathrm{T}_i) \right) \quad .$$

Table 5.1: Images available for the point clouds.

Data Set	Image Resolution	Camera	Num. Images	Num. Textures
Arch	3888 × 2592	EOS 400D	25	10
Cemetery	3264 × 2448	iPhone 5	14	9
Fountain	3264 × 2448	iPhone 5	30	11
Statues	3888 × 2592	EOS 400D	9	5

5.4 RESULTS

The reconstruction experiments with the Arch, Cemetery, Fountain, and Statues data sets that were performed earlier (Sec. 4.6) were repeated using the texture assignment extension of SGNG presented above. Each point cloud was created from photos by a combination of *VisualSFM* [130] and *CMVS/PMVS* [54]. Thus, the visibility information for the input points in the original images is readily available. Tab. 5.1 lists further details for the images available with the input point clouds. Fig. 5.2, 5.3 present thumbnails of the images as an overview. The photos were taken in Paris, France, with a Canon® Digital Rebel XTi (EOS 400D) DSLR or with the built-in camera of an Apple® iPhone® 5.

Visual quality

SGNG successfully assigns suitable textures to the constructed triangles with the extension presented in this chapter. Fig. 5.4–5.9 present the results. Original photos (Fig. 5.4(a), 5.5(a), 5.7(a), 5.8(a), 5.9(a)) that were not used as textures for the constructed triangle meshes are presented alongside the textured reconstructions (Fig. 5.4(b), 5.5(b), 5.7(b), 5.8(b), 5.9(b)). In order to alleviate visual evaluation the virtual camera used for rendering the reconstruction was aligned with and matched to the original camera used for taking the photo. Two different pairs of photo and reconstruction are presented for the Fountain data set due to the size of the original object. Since the original photos were leveled only manually, slight color and exposure mismatches are noticeable on the textured triangle meshes. However, no geometric misalignments or distortions are visible. Even the inscription on the tomb of the Cemetery data set is clearly legible (Fig. 5.6(b)).

Occlusion artifacts

SGNG successfully reduces the number of occlusion artifacts to a minimum by taking visibility into account. Textures are selected in such a way that the people walking in front of the Arch and the Fountain and that are visible in the original photos (Fig. 5.2(a), 5.3(a)) are not visible in the textures assigned to the constructed meshes (Fig. 5.4(b), 5.7(b), 5.8(b)). Static occluders are also handled correctly by SGNG. Fig. 5.6 presents the effect of learning the visibility showing close-ups of the textured reconstructions for the Cemetery data set. If only the most perpendicular view is used to select the textures, occlusion artifacts become clearly apparent (Fig. 5.6(a)): The blue

Table 5.2: Running times for SGNG without and with textures.

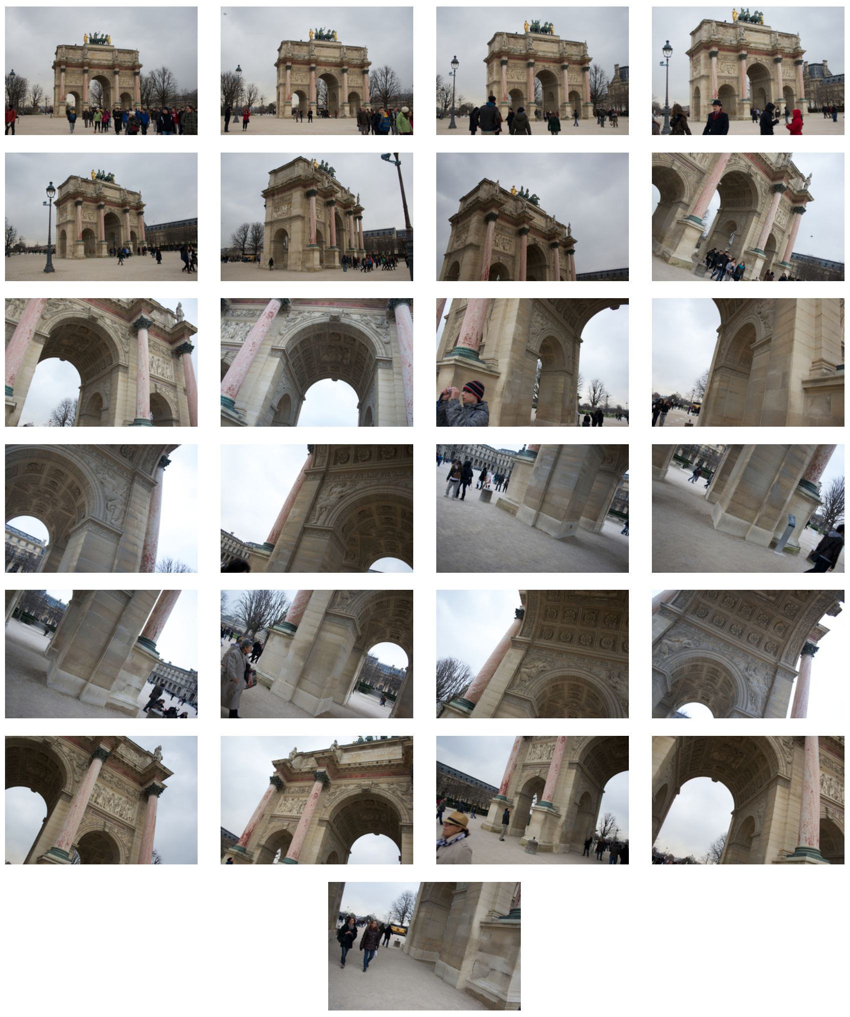
	Arch	Cemetery
Points (iterative steps)	407 229 (10 St.)	380 840 (9 St.)
Vertices	101 808	95 210
Time (without textures) / s	69.2	53.8
Time (with textures) / s	89.1	72.1
Overhead / s	19.9	18.3
Relative Overhead 10^{-5} s	2.0	2.1

	Fountain	Statues
Points (iterative steps)	567 920 (16 St.)	497 489 (4 St.)
Vertices	140 000	124 373
Time (without textures) / s	99.4	74.5
Time (with textures) / s	132.3	90.9
Overhead / s	32.9	16.4
Relative Overhead / 10^{-5} s	2.1	2.6

shore and the pillars that are supporting the roof of the tomb are projected onto the tomb. By also learning the visibility information from the input points the number of these artifacts is reduced to a minimum (Fig. 5.6(b)): No details on the tomb are covered by projected images of the shore or the pillars. The same applies to the two statues in front of the Fountain (Fig. 5.7(a), 5.8(a)) that are not projected to the back. Instead, SGNG correctly selects those images as textures that show the unoccluded base of the Fountain (Fig. 5.7(b), 5.8(b)).

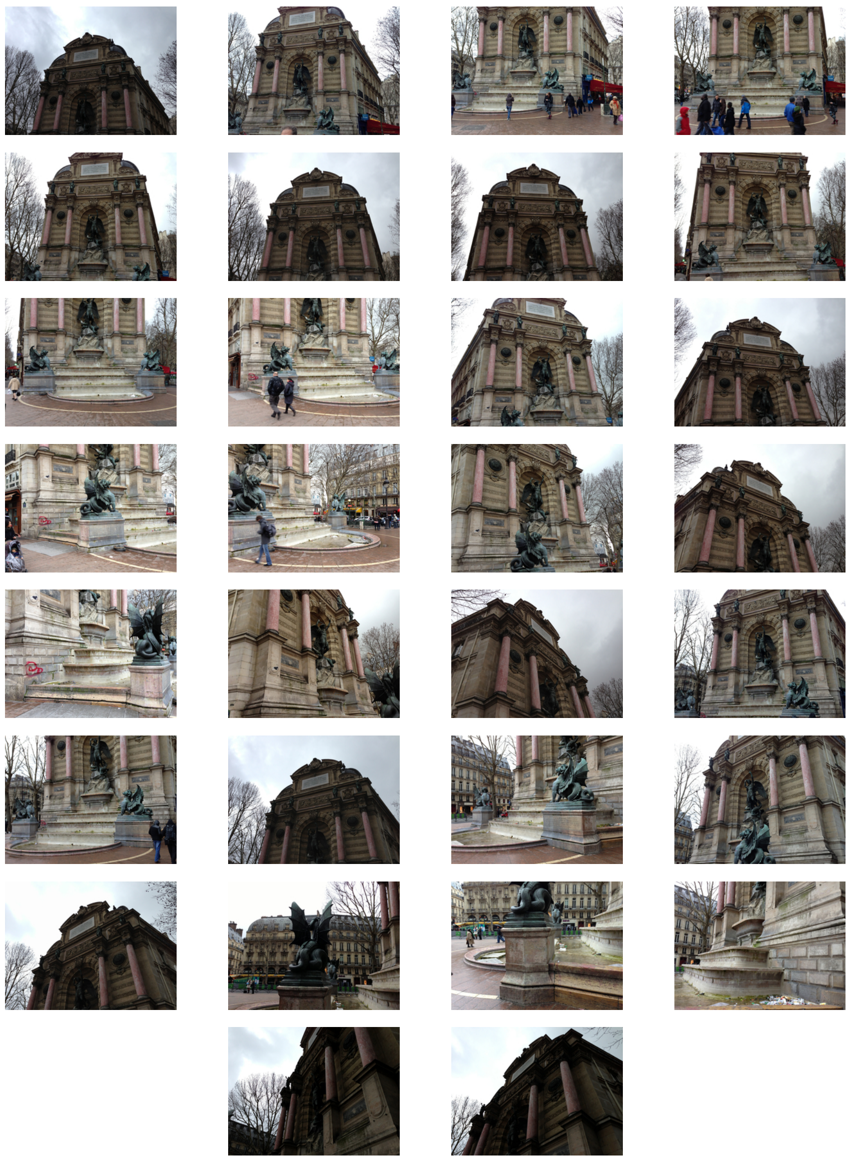
Overhead The texturing extension introduces an overhead to SGNG. Each vertex needs to keep track of its visibility in the input images. Thus, more memory is required. The running time overhead was evaluated explicitly. As in the experiments for SGNG (Sec. 4.6) the median running times of ten executions were determined. Learning the texture assignment takes additional 16.4 s to 32.9 s for the tested data sets. Tab. 5.2 presents the results. For a better overview some of the numbers of Tab. 4.4 are repeated.

Taking a closer look at the running times suggests that the overhead is related to the number of constructed vertices and the number of images used as textures (Tab. 5.1). Thus, the overhead appears to be related to the amount of additional data to be maintained and learned in order to enable texture assignment. As an indicator for this relationship, the bottom row of Tab. 5.2 presents the relative overhead, i.e., the difference of the running times divided by the product of the number of constructed vertices and the number of images used for texturing (Tab. 5.1). For all tested data sets the values of the relative overhead are in the same order of magnitude. However, a closer examination is deferred to future work.


(a) Arch

(b) Cemetery

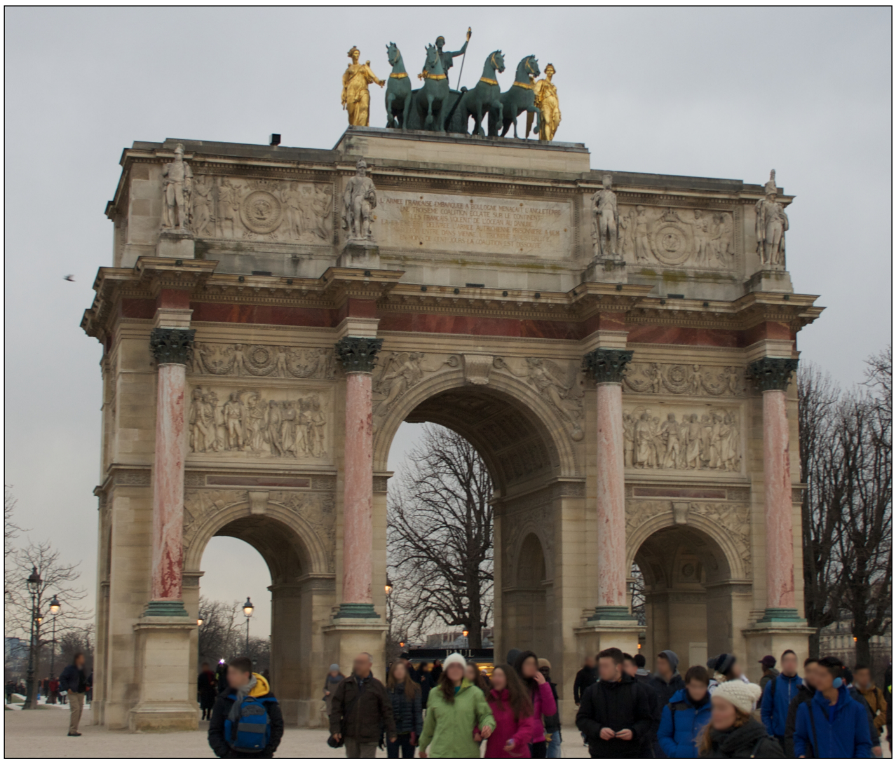
Figure 5.2: Thumbnails of the original photos – part 1.



(a) Fountain

(b) Statues

Figure 5.3: Thumbnails of the original photos – part 2.


(a) Photo that was not used for texturing.

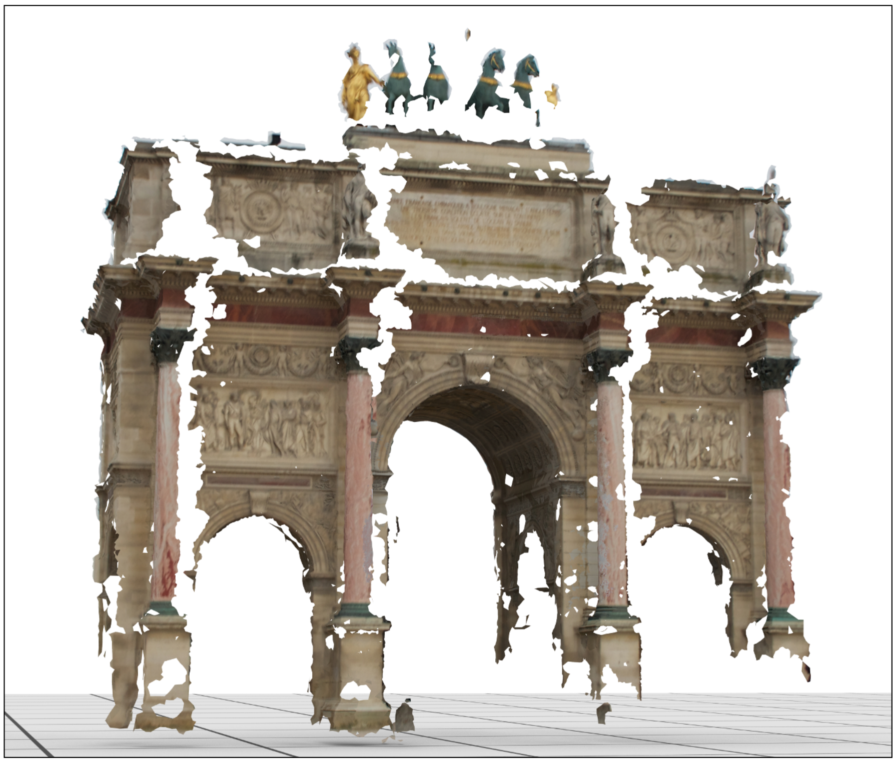
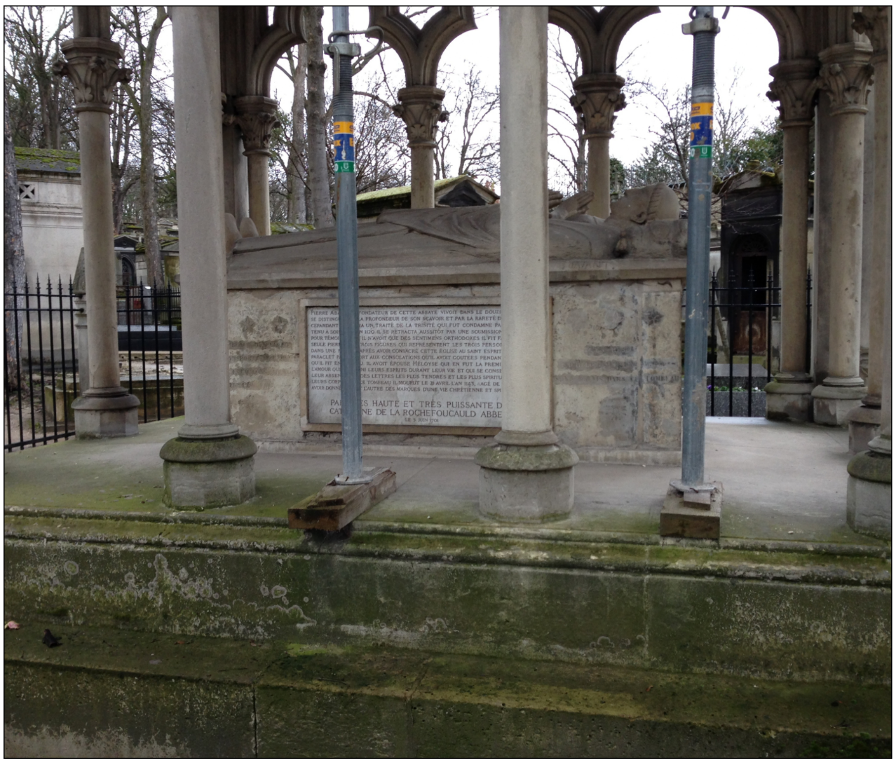

(b) Textured SGNG reconstruction.

Figure 5.4: Arch: Photo and textured SGNG reconstruction.


(a) Photo that was not used for texturing.

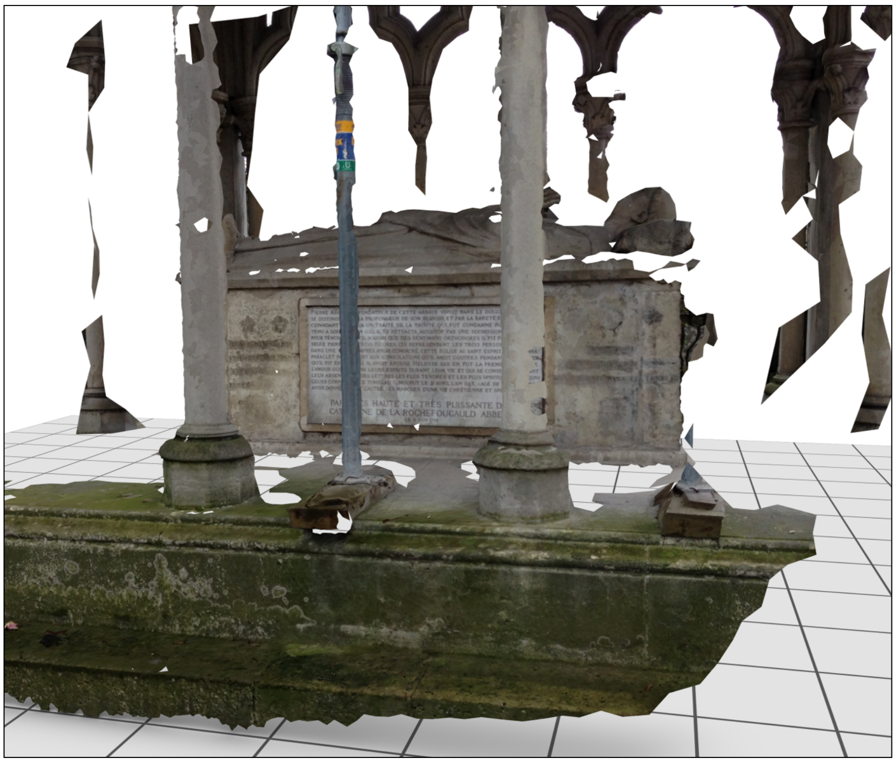
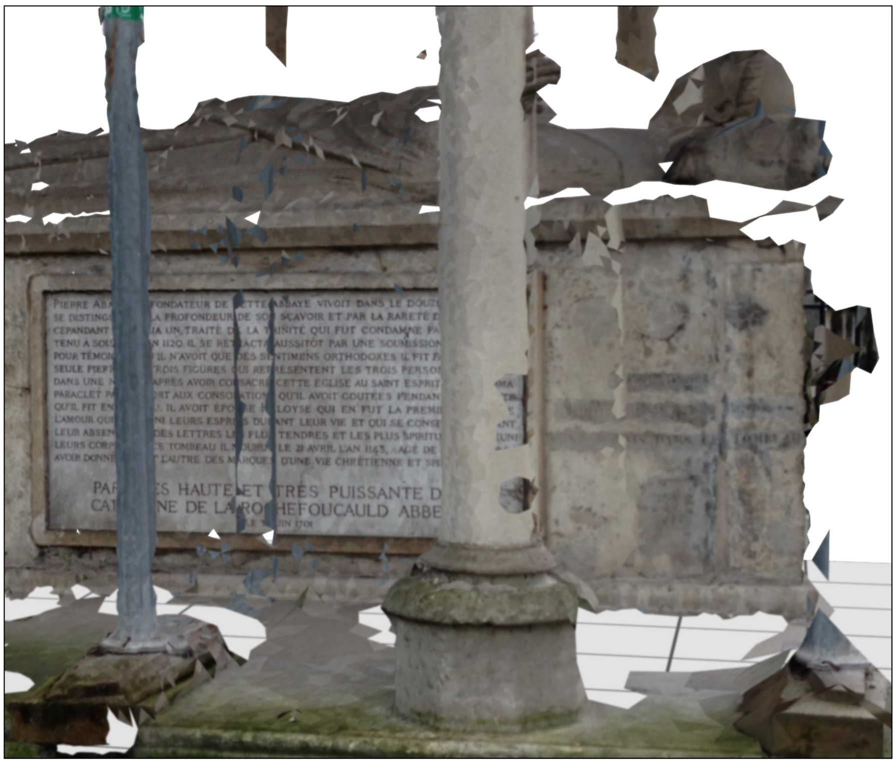
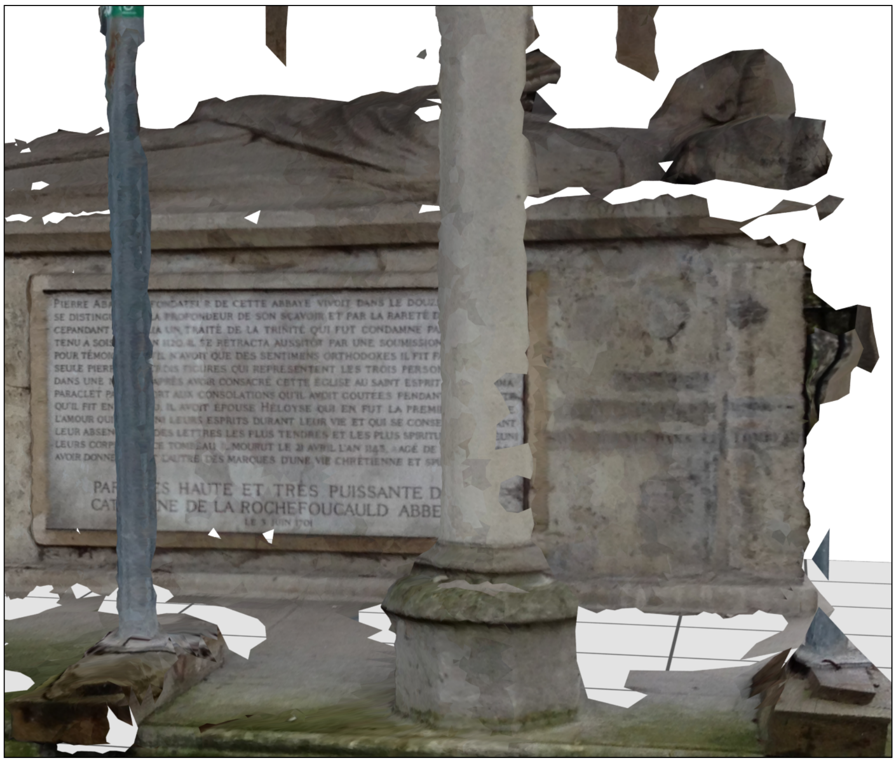

(b) Textured SGNG reconstruction.

Figure 5.5: Cemetery: Photo and textured SGNG reconstruction.

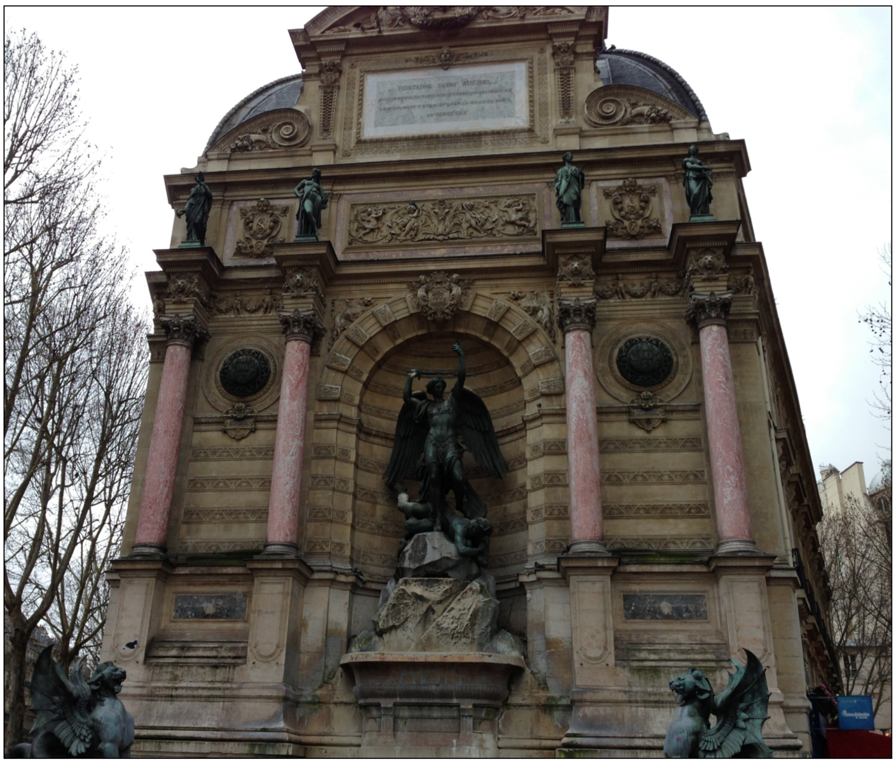


(a) Using only the most perpendicular view.



(b) Additionally using the learned visibility.

Figure 5.6: Cemetery: Close-ups of textured SGNG reconstructions.


(a) Photo that was not used for texturing.

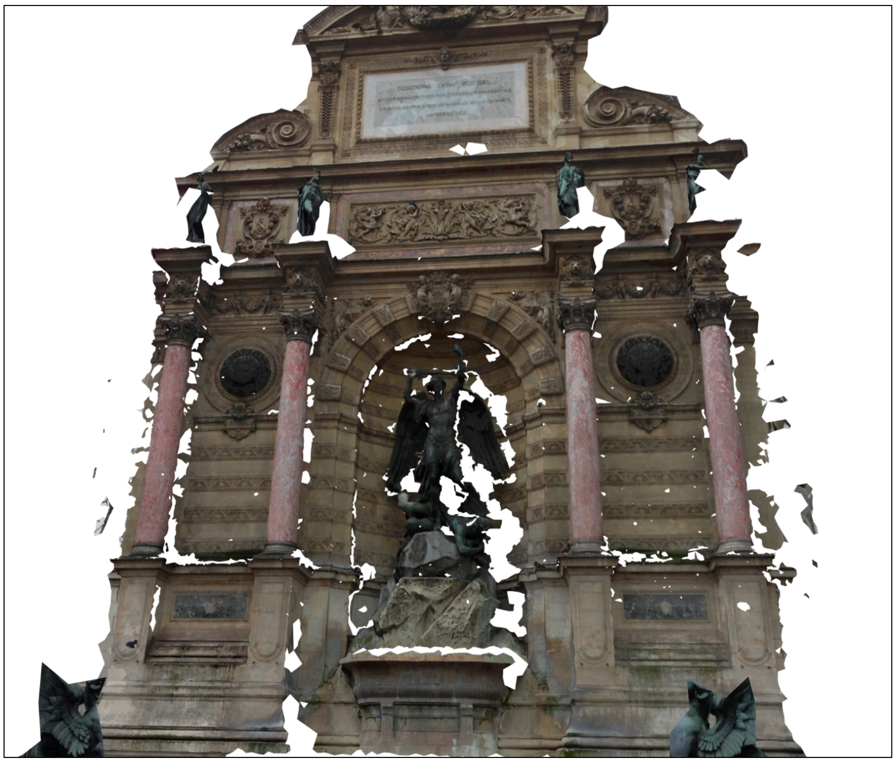
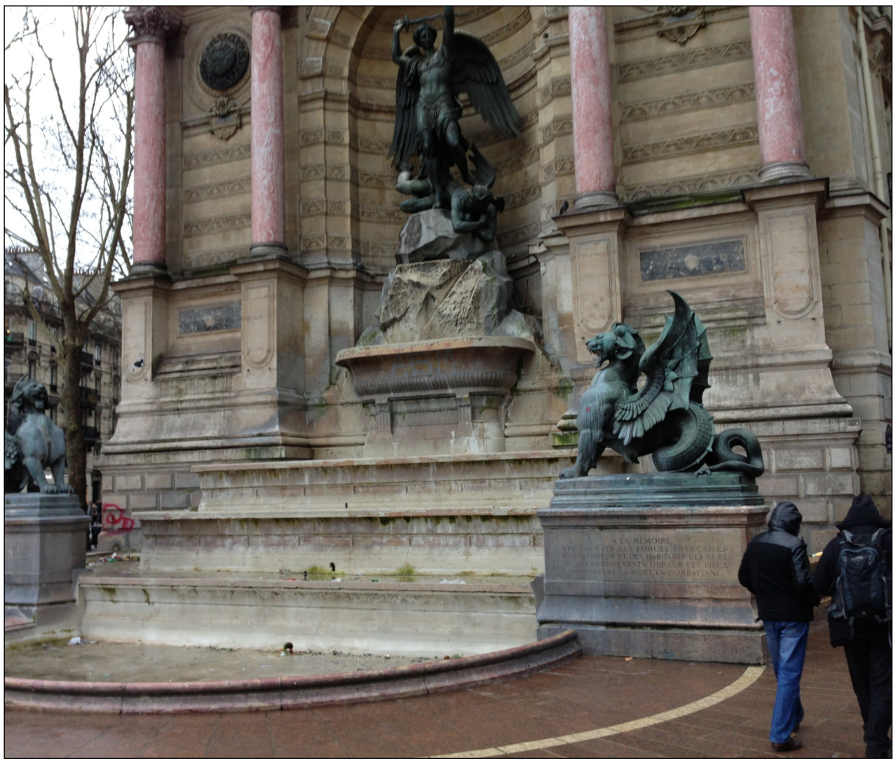
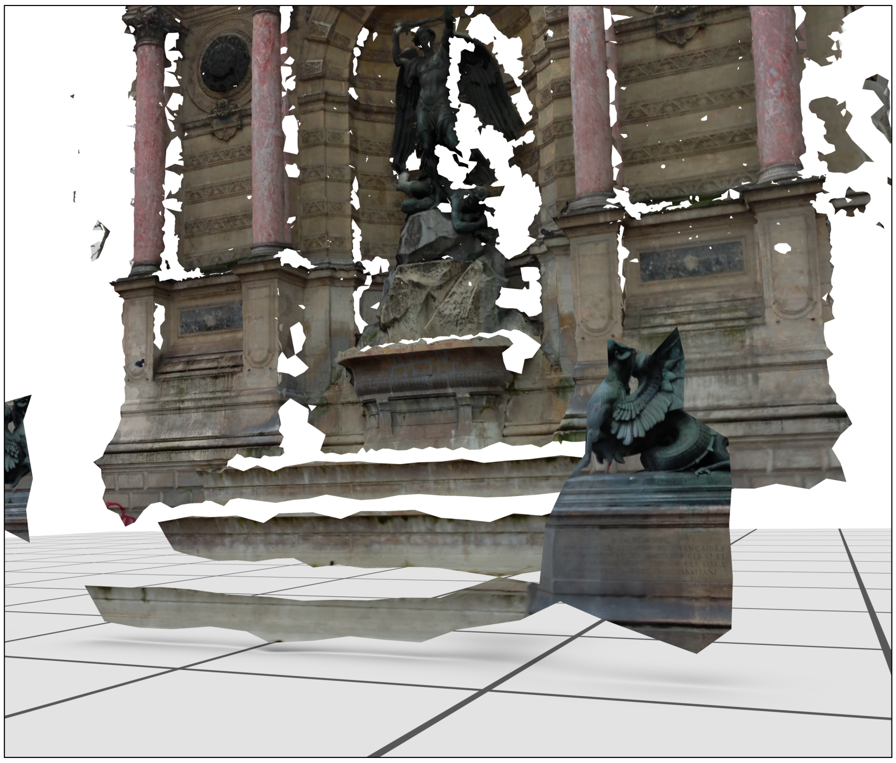

(b) Textured SGNG reconstruction.

Figure 5.7: Fountain: Photo and textured SGNG reconstruction.

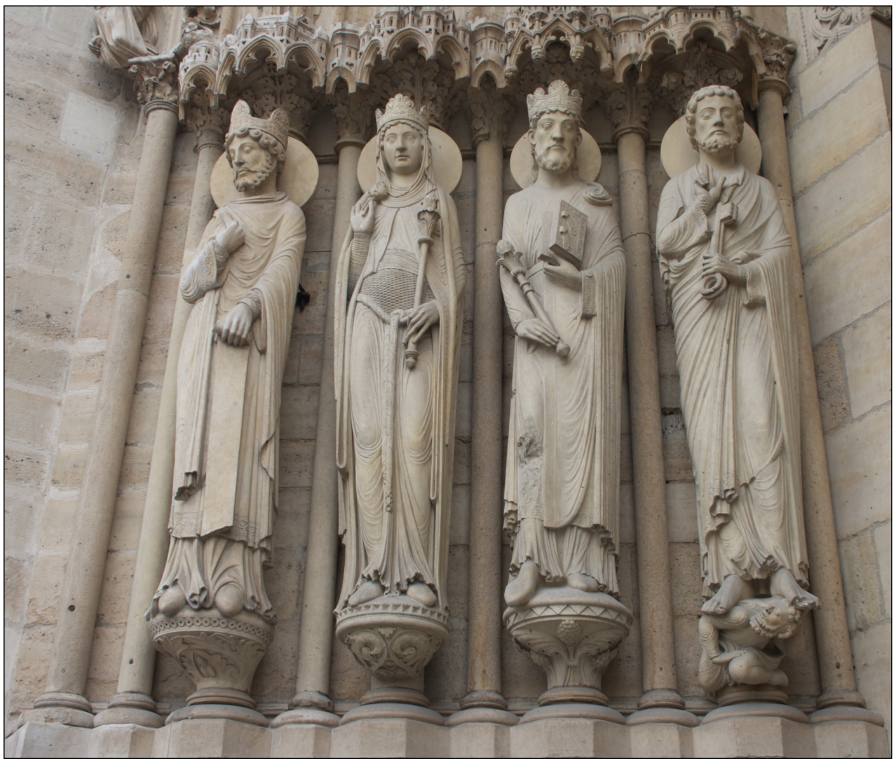


(a) Photo that was not used for texturing.

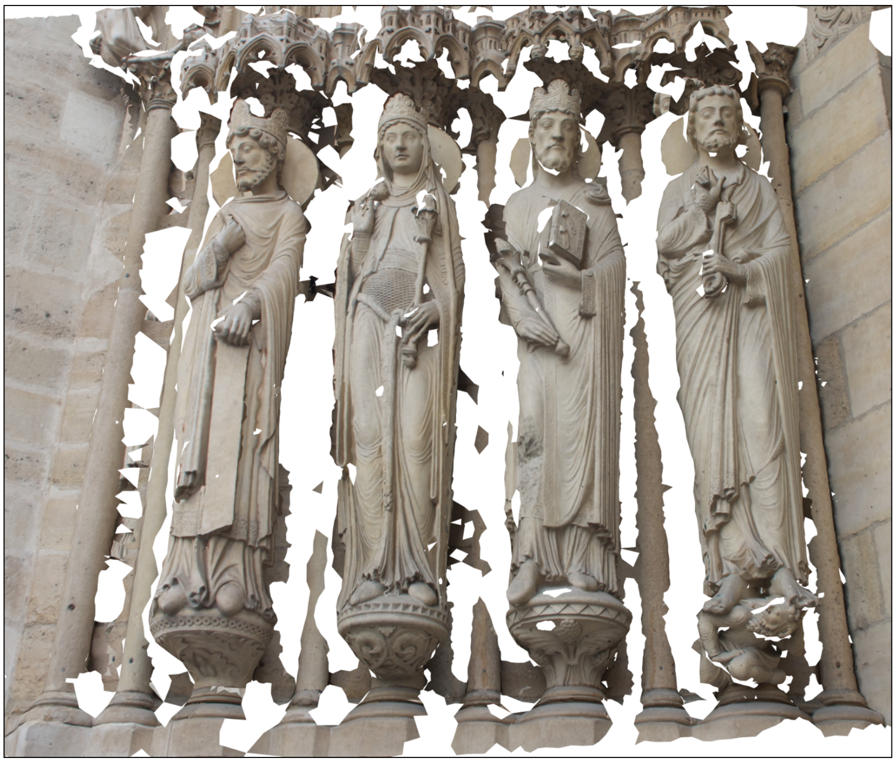


(b) Textured SGNG reconstruction.

Figure 5.8: Fountain: Photo and textured SGNG reconstruction.



(a) Photo that was not used for texturing.



(b) Textured SGNG reconstruction.

Figure 5.9: Statues: Photo and textured SGNG reconstruction.

5.5 DISCUSSION

The texturing extension that has been presented in this chapter enables SGNG to construct textured triangle meshes if the input points have been extracted from images, or if images are available that are registered to the points. That way further visual details are automatically added to the reconstructions that can be used for visualization at any time during the execution. Whenever new images are added during reconstruction, they are instantly considered as candidates for texturing the triangles. Once SGNG has learned that a new image provides a better view of a triangle, the image that has been used as a texture so far is replaced by the new one.

SGNG successfully reduces the number of occlusion artifacts to a minimum by learning the visibility information that is implicitly encoded in the input points. That way, no separate techniques for occlusion handling are required that might rely on accurately reconstructed occluders—a guarantee that is hard to fulfill in real-world applications, especially when using iterative and stochastic reconstruction approaches. Although the learned visibility leads to a fairly good texturing, some seams remain visible due to unmatched exposures and illuminations of the photos. Integrating existing leveling, blending and warping techniques will most likely alleviate this.

The current implementation of the texture assignment increases the time that is required to construct the triangle mesh from the input points. Evaluation suggests that this overhead is related to the amount of additional data that has to be maintained and learned. Improving data structures and memory management will most likely reduce this overhead and thus improve performance. However, even with the current implementation SGNG still outperforms screened Poisson surface reconstruction in terms of running times for the data sets with frequent updates during reconstruction. Only the Statues data set takes slightly longer, but with the benefit of included textures.

6 CONCLUSION AND FUTURE WORK

Surface-reconstructing growing neural gas (SGNG) has been presented in this dissertation as a technique for online surface reconstruction from unorganized point clouds that is based on an artificial neural network. The theoretical background for SGNG and its relation to other existing artificial neural networks that are used for surface reconstruction have been explained in detail. An in-depth analysis of the characteristics and shortcomings of prior algorithms has been used to derive the SGNG algorithm. Finally, a texturing extension for SGNG has been presented.

Unlike its predecessors SGNG expresses topological neighborhood via triangles. That way SGNG successfully constructs a triangle mesh entirely during online learning, reducing the number of untriangulated holes to a minimum. By taking geometric considerations into account, a mesh with fairly regular triangles is constructed that at first approximates an original object's shape and that is iteratively refined afterwards. An extensive evaluation has confirmed that SGNG improves significantly upon its predecessor and that it can compete with other state-of-the-art surface reconstruction algorithms.

The texturing extension enables SGNG to assign suitable textures to the constructed triangles if the input points have been extracted from images. That way further visual detail is automatically added to the reconstruction of the original object's surface. SGNG does not rely on the reconstructed geometry to determine visibility. Instead, visibility is learned directly from the information that is implicitly contained in the input data. Thus, the number of noticeable occlusion artifacts is reduced to a minimum. A visual evaluation has confirmed that SGNG yields good texturing quality. No severe geometric misalignments or distortions are visible.

SGNG allows for modifications of the input data at any time during reconstruction and instantly incorporates them without having to restart from scratch. Thus, SGNG is very well suited for an iterative pipeline, where data acquisition, reconstruction, and visualization are executed in parallel. Such a pipeline can be used for instance in emergency or disaster management scenarios, where it is crucial to get an immediate overview of the situation and to be able to add further data whenever needed. Especially during long running scenarios SGNG will play out its full strength.

The texturing extension does not yet use elaborate techniques for leveling or blending. Thus, some texture seams remain visible due to unmatched exposures and illuminations of the photos. Literature provides a wealth of suitable techniques that can be incorporated into SGNG. The descriptions and algorithmic details that have been presented in this dissertation will serve as a good basis for future work in this direction.

Memory management and data handling have turned out to be a bottleneck in both parts of the current implementation of SGNG: in reconstruction and in texturing. Running time in the former is dominated by the 2-nearest neighbor search. Overhead in terms of running time of the latter mostly depends on the number of constructed vertices and the number of textures in use. Some effort will be required to streamline the current implementation in this regard and thus to improve performance.

Currently, an adaptive octree is used in SGNG to find the two vertices that are closest to an input point. It is maintained as a separate data structure. Since the vertices of the constructed mesh are continuously moved during the iterations of SGNG reconstruction, many updates of the cells' boundaries are required as well as many vertex relocations between cells. A *split tree* has been developed and presented in this dissertation that serves as an efficient replacement for the octree in *growing cell structures* (GCS) by providing a hierarchy of bounding boxes. This hierarchy is automatically created during mesh refinement. It is very likely that the presented *split tree* can be adapted in such a way that it serves as an acceleration structure also for SGNG since the learning algorithms of SGNG and GCS are closely related. However, some improvements to the algorithm are required in order to create a balanced tree. These improvements and the adaptation of the *split tree* to SGNG have been beyond the scope of this work. They remain to be addressed in future work.

Feeding back the constructed textured triangle meshes into the point extraction process might be advantageous. That way, the geometric information reconstructed by SGNG may help the point extraction process generate a denser set of samples and register the images better. Furthermore, the idle times of one process could be used by the other. The combination of both would lead to an integrated, incremental coarse to fine approach, maybe with reduced processing power and memory requirements—perhaps even on mobile platforms.

Creating a dual representation of SGNG is an interesting task that presents itself for future work, since the current algorithm—like its predecessors—exhibits a systematic error: In SGNG the vertices' positions are optimized in such a way that their distance to the input points is minimized. Thus, the corresponding learning rules smooth sharp features if the constructed mesh contains too few vertices. Nevertheless, SGNG aims at reconstructing an original object's surface. Consequently, in contrast to the current approach, the surface elements, i.e., the constructed triangles, should be optimized in such a way that their distance to the input points is minimized. A straightforward integration of additional planar fitting made SGNG prone to noise, leading to many wrinkles. It is thus planned to investigate a more robust technique for dual SGNG surface reconstruction. The details presented in this dissertation will serve as a good starting point.

BIBLIOGRAPHY

[1] Ahmed Abdelhafiz. *Integrating Digital Photogrammetry and Terrestrial Laser Scanning*. PhD thesis, Technische Universität Carolo-Wilhelmina zu Braunschweig, Germany, 2009.

[2] Sameer Agarwal, Noah Snavely, Ian Simon, Steven M. Seitz, and Richard Szeliski. Building rome in a day. In *Proc. IEEE 12th Int. Conf. Comput. Vision*, pp. 72–79, 2009. doi: 10.1109/ICCV.2009.5459148.

[3] Motilal Agrawal, Kurt Konolige, and Morten Rufus Blas. Censure: Center surround extremas for realtime feature detection and matching. In *Proc. 10th European Conf. Comput. Vision*, pp. 102–115, 2008. doi: 10.1007/978-3-540-88693-8_8.

[4] Alexandre Alahi, Raphael Ortiz, and Pierre Vandergheynst. FREAK: Fast retina keypoint. In *Proc. IEEE Conf. Comput. Vision and Pattern Recognition*, pp. 510–517, 2012. doi: 10.1109/CVPR.2012.6247715.

[5] Nina Amenta, Marshall Bern, and Manolis Kamvysselis. A new voronoi-based surface reconstruction algorithm. In *Proc. SIGGRAPH 98, Annu. Conf. Series*, pp. 415–421, 1998. doi: 10.1145/280814.280947.

[6] Nina Amenta, Sunghee Choi, Tamal K. Dey, and Naveen Leekha. A simple algorithm for homeomorphic surface reconstruction. In *Proc. 16th Annu. Symp. Computational geometry*, pp. 213–222, 2000. doi: 10.1145/336154.336207.

[7] Nina Amenta, Sunghee Choi, and Ravi Krishna Kolluri. The power crust. In *Proc. 6th ACM Symp. Solid modeling and Applicat.*, pp. 249–266, 2001. doi: 10.1145/376957.376986.

[8] Hendrik Annuth and Christian-A. Bohn. Surface reconstruction with smart growing cells. In Dimitri Plemenos and Georgios Miaoulis, editors, *Intelligent Comput. Graph. 2010*, pp. 47–66. Springer, 2010. doi: 10.1007/978-3-642-15690-8_3.

[9] Hendrik Annuth and Christian-A. Bohn. Tumble tree – reducing complexity of the growing cells approach. In *Proc. 20th Int. Conf. Artificial Neural Networks*, pp. 228–236, 2010. doi: 10.1007/978-3-642-15825-4_28.

[10] Hendrik Annuth and Christian-A. Bohn. Approximation of geometric structures with growing cell structures and growing

neural gas - a performance comparison. In *Proc. 4th Int. Joint Conf. Computational Intell.*, pp. 552–557, 2012.

[11] Marco Attene, Marcel Campen, and Leif Kobbelt. Polygon mesh repairing: An application perspective. *ACM Comput. Surv.*, 45 (2):15:1–15:33, 2013. doi: 10.1145/2431211.2431214.

[12] Franz Aurenhammer. Power diagrams: Properties, algorithms and applications. *SIAM J. Computing*, 16(1):78–96, 1987. doi: 10.1137/0216006.

[13] Matthew Berger. A benchmark for surface reconstruction (svn revision 8), 2011. URL `http://www.cs.utah.edu/~bergerm/recon_bench/` (last checked 2015-06-17).

[14] Matthew Berger, Joshua A. Levine, Luis Gustavo Nonato, Gabriel Taubin, and Claudio T. Silva. A benchmark for surface reconstruction. *ACM Trans. Graph.*, 32(2):20:1–20:17, 2013. doi: 10.1145/2451236.2451246.

[15] Matthew Berger, Andrea Tagliasacchi, Lee M. Seversky, Pierre Alliez, Joshua A. Levine, Andrei Sharf, and Claudio T. Silva. State of the Art in Surface Reconstruction from Point Clouds. In *EG 2014 - STARs*, pp. 161–185, 2014. doi: 10.2312/egst.20141040.

[16] Fausto Bernardini, Joshua Mittleman, Holly Rushmeier, Cláudio Silva, and Gabriel Taubin. The ball-pivoting algorithm for surface reconstruction. *IEEE Trans. Vis. Comput. Graphics*, 5(4): 349–359, 1999. doi: 10.1109/2945.817351.

[17] Michael Birsak, Przemyslaw Musialski, Murat Arikan, and Michael Wimmer. Seamless texturing of archaeological data. In *Proc. Digital Heritage*, 2013.

[18] Jean-Daniel Boissonnat. Geometric structures for three-dimensional shape representation. *ACM Trans. Graph.*, 3(4): 266–286, 1984. doi: 10.1145/357346.357349.

[19] Jean-Daniel Boissonnat and Steve Oudot. Provably good sampling and meshing of surfaces. *Graph. Models*, 67(5):405–451, 2005. doi: 10.1016/j.gmod.2005.01.004.

[20] Matthew Bolitho, Michael Kazhdan, Randal Burns, and Hugues Hoppe. Multilevel Streaming for Out-of-Core Surface Reconstruction. In *Proc. 5th Eurographics Symp. Geometry Process.*, 2007. doi: 10.2312/SGP/SGP07/069-078.

[21] Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23(11):1222–1239, 2001. doi: 10.1109/34.969114.

[22] Fatih Calakli and Gabriel Taubin. Ssd-c: Smooth signed distance colored surface reconstruction. In John Dill, Rae Earnshaw, David Kasik, John Vince, and Pak Chung Wong, editors, *Expanding the Frontiers of Visual Analytics and Visualization*, pp. 323–338. Springer, 2012. doi: 10.1007/978-1-4471-2804-5_18.

[23] Marco Callieri, Paolo Cignoni, Massimiliano Corsini, and Roberto Scopigno. Masked photo blending: Mapping dense photographic data set on high-resolution sampled 3d models. *Computers & Graphics*, 32(3):464–473, 2008. doi: 10.1016/j.cag.2008.05.004.

[24] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. BRIEF: Binary robust independent elementary features. In *Proc. 11th European Conf. Comput. Vision*, 2010. doi: 10.1007/978-3-642-15561-1_56.

[25] Frédéric Cazals and Joachim Giesen. Delaunay triangulation based surface reconstruction. In Jean-Daniel Boissonnat and Monique Teillaud, editors, *Effective Computational Geometry for Curves and Surfaces*, pp. 231–276. Springer, 2006. doi: 10.1007/978-3-540-33259-6_6.

[26] Zhaolin Chen, Jun Zhou, Yisong Chen, and Guoping Wang. 3d texture mapping in multi-view reconstruction. In George Bebis, Richard Boyle, Bahram Parvin, Darko Koracin, Charless Fowlkes, Sen Wang, Min-Hyung Choi, Stephan Mantler, Jürgen Schulze, Daniel Acevedo, Klaus Mueller, and Michael Papka, editors, *Advances in Visual Computing*, pp. 359–371. Springer, 2012. doi: 10.1007/978-3-642-33179-4_35.

[27] Paolo Cignoni, Claudio Rocchini, and Roberto Scopigno. Metro: Measuring error on simplified surfaces. *Comput. Graph. Forum*, 17(2):167–174, 1998. doi: 10.1111/1467-8659.00236.

[28] Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, and Guido Ranzuglia. Meshlab: an open-source mesh processing tool. In *6th Eurographics Italian Chapter Conf.*, pp. 129–136, 2008. doi: 10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136.

[29] Massimiliano Corsini, Paolo Cignoni, and Roberto Scopigno. Efficient and flexible sampling with blue noise properties of triangular meshes. *IEEE Trans. Vis. Comput. Graphics*, 18(6):914–924, 2012. doi: 10.1109/TVCG.2012.34.

[30] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *Proc. SIGGRAPH 96, Annu. Conf. Series*, pp. 303–312, 1996. doi: 10.1145/237170.237269.

[31] Vilson Luiz DalleMole and Aluizio Fausto Ribeiro Araújo. The growing self-organizing surface map. In *IEEE Int. Joint Conference Neural Networks*, pp. 2061–2068, 2008. doi: 10.1109/IJCNN.2008.4634081.

[32] Vilson Luiz DalleMole and Aluizio Fausto Ribeiro Araújo. Growing self-organizing surface map: Learning a surface topology from a point cloud. *Neural Computation*, 22(3):689–729, 2010. doi: 10.1162/neco.2009.08-08-842.

[33] Vilson Luiz DalleMole, Renata L.M.E. do Rêgo, and Aluizio Fausto Ribeiro Araújo. The self-organizing approach for surface reconstruction from unstructured point clouds. In George K. Matsopoulos, editor, *Self-Organizing Maps*, chapter 11, pp. 167–188. InTech, 2010.

[34] Matteo Dellepiane, Ricardo Marroquim, Marco Callieri, Paolo Cignoni, and Roberto Scopigno. Flow-based local optimization for image-to-geometry projection. *IEEE Trans. Vis. Comput. Graphics*, 18(3):463–474, 2012. doi: 10.1109/TVCG.2011.75.

[35] Tamal K. Dey. *Curve and Surface Reconstruction: Algorithms with Mathematical Analysis*. Cambridge University Press, 2006.

[36] Tamal K. Dey and Samrat Goswami. Tight cocone: a water-tight surface reconstructor. In *Proc. 8th ACM Symp. Solid modeling and Applicat.*, pp. 127–134, 2003. doi: 10.1145/781606.781627.

[37] Tamal K. Dey and Samrat Goswami. Provable surface reconstruction from noisy samples. *Computational Geometry*, 35(1–2): 124–141, 2006. doi: 10.1016/j.comgeo.2005.10.006.

[38] Tamal K. Dey and Lei Wang. Voronoi-based feature curves extraction for sampled singular surfaces. *Computers & Graphics*, 37(6):659–668, 2013. doi: 10.1016/j.cag.2013.05.014.

[39] Renata L.M.E. do Rêgo. *Mapas Auto-organizáveis com Estrutura Variante do Tempo para Reconstrução de Superfícies*. PhD thesis, Universidade Federal de Pernambuco, Recife, Brazil, 2013.

[40] Renata L.M.E. do Rêgo and Aluizio Fausto Ribeiro Araujo. A surface reconstruction method based on self-organizing maps and intrinsic delaunay triangulation. In *2010 Int. Joint Conf. Neural Networks*, pp. 1–8, 2010. doi: 10.1109/IJCNN.2010.5596929.

[41] Renata L.M.E. do Rêgo, Aluizio Fausto Ribeiro Araújo, and Fernando Buarque de Lima Neto. Growing self-reconstruction maps. *IEEE Trans. Neural Netw.*, 21(2):211–223, 2010. doi: 10.1109/TNN.2009.2035312.

[42] Herbert Edelsbrunner. Power diagrams. In *Algorithms in Combinatorial Geometry*, chapter 13.6, pp. 327–328. Springer, 1987. doi: 10.1007/978-3-642-61568-9.

[43] Herbert Edelsbrunner and Ernst P. Mücke. Three-dimensional alpha shapes. *ACM Trans. Graph.*, 13(1):43–72, 1994. doi: 10.1145/174462.156635.

[44] Herbert Edelsbrunner, David G. Kirkpatrick, and Raimund Seidel. On the shape of a set of points in the plane. *IEEE Trans. Inf. Theory*, 29(4):551–559, 1983. doi: 10.1109/TIT.1983.1056714.

[45] Ed Erwin, Klaus Obermayer, and Klaus Schulten. Self-organizing maps: ordering, convergence properties and energy functions. *Biological Cybern.*, 67(1):47–55, 1992. doi: 10.1007/BF00201801.

[46] Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Comm. ACM*, 24(6):381–395, 1981. doi: 10.1145/358669.358692.

[47] Daniel Fišer, Jan Faigl, and Miroslav Kulich. Growing neural gas efficiently. *Neurocomputing*, 104:72–82, 2013. doi: 10.1016/j.neucom.2012.10.004.

[48] Jan-Michael Frahm, Pierre Fite-Georgel, David Gallup, Tim Johnson, Rahul Raguram, Changchang Wu, Yi-Hung Jen, Enrique Dunn, Brian Clipp, Svetlana Lazebnik, and Marc Pollefeys. Building rome on a cloudless day. In *Proc. 11th European Conf. Comput. Vision*, pp. 368–381, 2010. doi: 10.1007/978-3-642-15561-1_27.

[49] Bernd Fritzke. *Wachsende Zellstrukturen - ein selbstorganisierendes neuronales Netzwerkmodell*. Dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany, 1992.

[50] Bernd Fritzke. Supervised learning with growing cell structures. In *Proc. Conf. Advances in Neural Inf. Process. Syst. 6*, pp. 255–262, 1993.

[51] Bernd Fritzke. Growing cell structures—a self-organizing network for unsupervised and supervised learning. *Neural Networks*, 7(9):1441 – 1460, 1994. doi: http://dx.doi.org/10.1016/0893-6080(94)90091-4.

[52] Bernd Fritzke. A growing neural gas network learns topologies. In *Proc. Conf. Advances in Neural Inf. Process. Syst. 7*, pp. 625–632, 1995.

[53] Bernd Fritzke. *Vektorbasierte Neuronale Netzte [Vector-Based Neural Networks]*. Professorial dissertation, Friedrich-Alexander-Univ. Erlangen-Nuernberg, Erlangen, Germany, 1998.

[54] Yasutaka Furukawa and Jean Ponce. Accurate, dense, and robust multi-view stereopsis. In *IEEE Conf. Comput. Vision and Pattern Recognition*, pp. 1–8, 2007. doi: 10.1109/CVPR.2007.383246.

[55] Ran Gal, Ariel Shamir, Tal Hassner, Mark Pauly, and Daniel Cohen-Or. Surface reconstruction using local shape priors. In *Proc. 5th Eurographics Symp. Geometry Process.*, pp. 253–262, 2007.

[56] Ran Gal, Yonatan Wexler, Eyal Ofek, Hugues Hoppe, and Daniel Cohen-Or. Seamless montage for texturing models. *Comput. Graph. Forum*, 29(2):479–486, 2010.

[57] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *Proc. SIGGRAPH 97, Annu. Conf. Series*, pp. 209–216, 1997. doi: 10.1145/258734.258849.

[58] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.*, 7(5):14–20, 1987. doi: 10.1109/MCG.1987.276983.

[59] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. *Comput. Graph.*, 18(3):213–222, 1984. doi: 10.1145/964965.808601.

[60] Miklos Hoffmann and Lajos Varady. Free-form surfaces for scattered data by neural networks. *J. Geometry and Graph.*, 2(1): 1–6, 1998.

[61] Yaron Holdstein and Anath Fischer. Three-dimensional surface reconstruction using meshing growing neural gas (mgng). *The Visual Computer*, 24(4):295–302, 2008. doi: 10.1007/s00371-007-0202-z.

[62] Hugues Hoppe. Progressive meshes. In *Proc. SIGGRAPH 96, Annu. Conf. Series*, pp. 99–108, 1996. doi: http://doi.acm.org/10.1145/237170.237216.

[63] Hugues Hoppe. New quadric metric for simplifying meshes with appearance attributes. In *Proc 10th IEEE Visualization Conf.*, pp. 59–66, 1999. doi: 10.1109/VISUAL.1999.809869.

[64] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Surface reconstruction from unorganized points. *Comput. Graph.*, 26(2):71–78, 1992. doi: 10.1145/133994.134011.

[65] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh Optimization. In *Proc. SIGGRAPH 93, Annu. Conf. Series*, pp. 19–26, 1993. doi: 10.1145/166117.166119.

[66] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh Optimization. Tech. rep., Dept. Computer Science & Engineering, Univ. Washington, 1993.

[67] Alexander Hornung and Leif Kobbelt. Robust reconstruction of watertight 3d models from non-uniformly sampled point clouds without normal information. In *Proc. 4th Eurographics Symp. Geometry Process.*, pp. 41–50, 2006. doi: 10.2312/SGP/SGP06/041-050.

[68] Alexander Hornung and Leif Kobbelt. Hierarchical volumetric multi-view stereo reconstruction of manifold surfaces based on dual graph embedding. In *Proc. 2006 IEEE Comput. Soc. Conf. Comput. Vision and Pattern Recognition*, pp. 503–510, 2006. doi: 10.1109/CVPR.2006.135.

[69] Liang Hu, Pedro V. Sander, and Hugues Hoppe. Parallel view-dependent refinement of progressive meshes. In *Proc. 2009 Symp. Interactive 3D Graph. and Games*, pp. 169–176, 2009. doi: 10.1145/1507149.1507177.

[70] Ioannis Ivrissimtzis, Won-Ki Jeong, and Hans-Peter Seidel. Using growing cell structures for surface reconstruction. In *Proc. Int. Conf. Shape Modeling and Applicat.*, pp. 78–86, 2003. doi: 10.1109/SMI.2003.1199604.

[71] Ioannis Ivrissimtzis, Won-Ki Jeong, and Hans-Peter Seidel. Neural meshes: Statistical learning methods in surface reconstruction. Tech. Rep. MPI–I–2003–4–007, Max-Planck-Institut für Informatik, 2003.

[72] Ioannis Ivrissimtzis, Won-Ki Jeong, Seungyong Lee, Yunjin Lee, and Hans-Peter Seidel. Surface reconstruction based on neural meshes. In *Proc. Math. Methods for Curves and Surfaces*, pp. 223–242, 2004.

[73] Ioannis Ivrissimtzis, Won-Ki Jeong, Seungyong Lee, Yunjin Lee, and Hans-Peter Seidel. Neural meshes: Surface reconstruction with a learning algorithm. Tech. Rep. MPI–I–2004–4–005, Max-Planck-Institut für Informatik, 2004.

[74] Ioannis Ivrissimtzis, Yunjin Lee, Seungyong Lee, Won-Ki Jeong, and Hans-Peter Seidel. Neural mesh ensembles. In *Proc. 2nd Int. symp. 3D Data Process., Visualization, and Transmission*, pp. 308–315, 2004. doi: 10.1109/TDPVT.2004.1335216.

[75] Michael Kazhdan. Reconstruction of solid models from oriented point sets. In *Proc. 3rd Eurographics Symp. Geometry Process.*, 2005. doi: 10.2312/SGP/SGP05/073-082.

[76] Michael Kazhdan and Matthew Bolitho. Poisson surface reconstruction (version 2), 2011. URL `http://www.cs.jhu.edu/~misha/Code/PoissonRecon/Version2/` (last checked 2015-06-17).

[77] Michael Kazhdan and Matthew Bolitho. Screened poisson surface reconstruction (version 6.13), 2014. URL `http://www.cs.jhu.edu/~misha/Code/PoissonRecon/Version6.13` (last checked 2015-06-17).

[78] Michael Kazhdan and Hugues Hoppe. Screened poisson surface reconstruction. *ACM Trans. Graph.*, 32(3):29:1–29:13, 2013. doi: 10.1145/2487228.2487237.

[79] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson Surface Reconstruction. In *Proc. 4th Eurographics Symp. Geometry Process.*, pp. 61–70, 2006. doi: 10.2312/SGP/SGP06/061-070.

[80] Donald E. Knuth. Literate programming. *The Comput. J.*, 27(2): 97–111, 1984. doi: 10.1093/comjnl/27.2.97.

[81] Teuvo Kohonen. Automatic formation of topological maps of patterns in a self-organizing system. In *Proc. 2nd Scand. Conf. Image Anal.*, 1981.

[82] Teuvo Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybern.*, 43(1):59–69, 1982. doi: 10.1007/BF00337288.

[83] Teuvo Kohonen. The self-organizing map. *Proc. IEEE*, 78(9): 1464–1480, 1990. doi: 10.1109/5.58325.

[84] Ravikrishna Kolluri, Jonathan Richard Shewchuk, and James F. O'Brien. Spectral Surface Reconstruction From Noisy Point Clouds. In *Proc. 2nd Eurographics Symp. Geometry Process.*, pp. 11–22, 2004. doi: 10.2312/SGP/SGP04/011-022.

[85] Victor Lempitsky and Denis Ivanov. Seamless mosaicing of image-based texture maps. In *IEEE Conf. Comput. Vision and Pattern Recognition*, pp. 1–6, 2007. doi: 10.1109/CVPR.2007.383078.

[86] Peter Lindstrom and Greg Turk. Fast and memory efficient polygonal simplification. In *Proc. Conf. Visualization '98*, pp. 279–286, 1998. doi: 10.1109/VISUAL.1998.745314.

[87] Stuart P. Lloyd. Least squares quantization in pcm. *IEEE Trans. Inf. Theory*, 28(2):129–137, 1982. doi: 10.1109/TIT.1982.1056489.

[88] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Comput. Graph.*, 21(4):163–169, 1987. doi: 10.1145/37402.37422.

[89] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, 2004. doi: 10.1023/B:VISI.0000029664.99615.94.

[90] James B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. 5th Berkeley Symp. Math. Stat. and Probability*, pp. 281–297, 1967.

[91] Thomas Martinetz. Competitive hebbian learning rule forms perfectly topology preserving maps. In *Proc. Int. Conf. Artificial Neural Networks*, pp. 427–434, 1993. doi: 10.1007/978-1-4471-2063-6_104.

[92] Thomas Martinetz and Klaus Schulten. A "neural-gas" network learns topologies. *Artificial Neural Networks*, 1:397–402, 1991.

[93] Thomas Martinetz and Klaus Schulten. Topology Representing Networks. *Neural Networks*, 7(3):507–522, 1994. doi: 10.1016/0893-6080(94)90109-0.

[94] Thomas Martinetz, Stanislav G. Berkovich, and Klaus Schulten. "neural-gas" network for vector quantization and its application to time-series prediction. *IEEE Trans. Neural Netw.*, 4(4):558–569, 1993. doi: 10.1109/72.238311.

[95] Markus Melato, Barbara Hammer, and Kai Hormann. Neural gas for surface reconstruction. Tech. Rep. IfI-07-08, Dept. Informatics, Clausthal Univ. of Technology, Clausthal, Germany, 2007.

[96] Microsoft®. MSDN®: ProgressiveMesh class. URL `https://msdn.microsoft.com/en-us/library/windows/desktop/bb281243(v=vs.85).aspx` (last checked 2015-06-17).

[97] Patrick Mullen, Fernando De Goes, Mathieu Desbrun, David Cohen-Steiner, and Pierre Alliez. Signing the Unsigned: Robust Surface Reconstruction from Raw Pointsets. *Comput. Graph. Forum*, 29(5):1733–1741, 2010. doi: 10.1111/j.1467-8659.2010.01782.x.

[98] Przemyslaw Musialski, Christian Luksch, Michael Schwärzler, Matthias Buchetics, Stefan Maierhofer, and Werner Purgathofer. Interactive multi-view façade image editing. In *Vision, Modeling and Visualization Workshop 2010*, pp. 131–138, 2010. doi: 10.2312/PE/VMV/VMV10/131-138.

[99] Przemyslaw Musialski, Peter Wonka, Daniel G. Aliaga, Michael Wimmer, Luc J. van Gool, and Werner Purgathofer. A survey of urban reconstruction. *Comput. Graph. Forum*, 32(6):146–177, 2013. doi: 10.1111/cgf.12077.

[100] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohli, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *Proc. 10th IEEE Int. Symp. Mixed and Augmented Reality*, pp. 127–136, 2011. doi: 10.1109/ISMAR.2011.6092378.

[101] Sergio Orts-Escolano, Jose Garcia-Rodriguez, Vicente Moreli, Miguel Cazorla, and Juan Manuel Garcia-Chamizo. 3d colour object reconstruction based on growing neural gas. In *Int. Joint Conf. Neural Networks*, pp. 1474–1481, 2014. doi: 10.1109/IJCNN.2014.6889546.

[102] Sergio Orts-Escolano, Jose Garcia-Rodriguez, Vicente Morell, Miguel Cazorla, JoseAntonioSerra Perez, and Alberto Garcia-Garcia. 3d surface reconstruction of noisy point clouds using growing neural gas: 3d object/scene reconstruction. *Neural Process. Lett.*, pp. 1–23, 2015. doi: 10.1007/s11063-015-9421-x.

[103] Chao Peng, Seung Park, Yong Cao, and Jie Tian. A real-time system for crowd rendering: Parallel lod and texture-preserving approach on gpu. In Jan Allbeck and Petros Faloutsos, editors, *Motion in Games*, pp. 27–38. Springer, 2011. doi: 10.1007/978-3-642-25090-3_3.

[104] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., 2nd edition, 2010.

[105] Sebastian Rohde, Niklas Goddemeier, Christian Wietfeld, Frank Steinicke, Klaus Hinrichs, Tobias Ostermann, Johanna Holsten, and Dieter Moormann. Avigle: A system of systems concept for an avionic digital service platform based on micro unmanned aerial vehicles. In *Proc. IEEE Int. Conf. Syst. Man and Cybern.*, pp. 459–466, 2010. doi: 10.1109/ICSMC.2010.5641767.

[106] Jan Roters and Xiaoyi Jiang. Festgpu: a framework for fast robust estimation on gpu. *J. Real-Time Image Process.*, pp. 1–14, 2014. doi: 10.1007/s11554-014-0439-5.

[107] Jan Roters, Frank Steinicke, and Klaus H. Hinrichs. Quasi-real-time 3d reconstruction from low-altitude aerial images. In *Proc. 40th Urban Data Manage. Soc. Symp.*, pp. 231–241, 2011. doi: 10.1201/b11647-24.

[108] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proc. SIGGRAPH 00, Annu. Conf. Series*, pp. 343–352, 2000. doi: 10.1145/344779.344940.

[109] Szymon Rusinkiewicz, Olaf Hall-Holt, and Marc Levoy. Real-time 3d model acquisition. *ACM Trans. Graph.*, 21(3):438–446, 2002. doi: 10.1145/566654.566600.

[110] Waqar Saleem. A flexible framework for learning-based surface reconstruction. Master's thesis, Univ. of Saarland, Saarbruecken, Germany, 2004.

[111] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., 2005.

[112] Yoram Singer and Manfred K. Warmuth. Batch and on-line parameter estimation of gaussian mixtures based on the joint entropy. In *Proc. Advances in Neural Inf. Process. Syst. 11*, pp. 578–584, 1998.

[113] Sudipta N. Sinha, Drew Steedly, Richard Szeliski, Maneesh Agrawala, and Marc Pollefeys. Interactive 3d architectural modeling from unordered photo collections. *ACM Trans. Graph.*, 27 (5):159:1–159:10, 2008. doi: 10.1145/1409060.1409112.

[114] Stanford Computer Graphics Laboratory. The stanford 3d scanning repository. URL `http://graphics.stanford.edu/data/3Dscanrep/` (last checked 2015-06-17).

[115] Sven Strothoff, Frank Steinicke, Dirk Feldmann, Jan Roters, Klaus Hinrichs, Tom Vierjahn, Markus Dunkel, and Sina Mostafawy. A virtual reality-based simulator for avionic digital service platforms. In *Proc. Joint Virtual Reality Conf. EGVE – EuroVR – VEC / Ind. Track*, 2010.

[116] Sven Strothoff, Dirk Feldmann, Frank Steinicke, Tom Vierjahn, and Sina Mostafawy. Interactive generation of virtual environments using MUAVs. In *Proc. IEEE Int. Symp. VR Innovations*, pp. 89 – 96, 2011. doi: 10.1109/ISVRI.2011.5759608.

[117] Gabriel Taubin. A signal processing approach to fair surface design. In *Proc. of SIGGRAPH 95, Annu. Conf. Series*, pp. 351–358, 1995. doi: 10.1145/218380.218473.

[118] Engin Tola, Vincent Lepetit, and Pascal Fua. Daisy: An efficient dense descriptor applied to wide baseline stereo. *IEEE Trans. Pattern Anal. Mach. Intell.*, 32(5):815–830, 2010. doi: 10.1109/TPAMI.2009.77.

[119] Lajos Várady, Miklós Hoffmann, and Emőd Kovács. Improved free-form modelling of scattered data by dynamic neural networks. *J. Geometry and Graph.*, 3:177–183, 1999.

[120] Tom Vierjahn and Klaus Hinrichs. Surface-reconstructing growing neural gas: A method for online construction of textured triangle meshes. *Computers & Graphics*, pp. –, 2015. doi: 10.1016/j.cag.2015.05.016.

[121] Tom Vierjahn, Guido Lorenz, Sina Mostafawy, and Klaus Hinrichs. Growing cell structures learning a progressive mesh during surface reconstruction – a top-down approach. In *Eurographics 2012 - Short Papers*, 2012. doi: 10.2312/conf/EG2012/short/029-032.

[122] Tom Vierjahn, Niklas Henrich, Klaus Hinrichs, and Sina Mostafawy. sgng: Online surface reconstruction based on growing neural gas. Tech. rep., Dept. Computer Science, Univ. Muenster, Germany, 2013.

[123] Tom Vierjahn, Jan Roters, Manuel Moser, Klaus Hinrichs, and Sina Mostafawy. Online reconstruction of textured triangle meshes from aerial images. In *1st Eurographics Workshop Urban Data Modelling and Visualisation*, pp. 1–4, 2013. doi: 10.2312/UDMV/UDMV13/001-004.

[124] Ruimin Wang, Zhouwang Yang, Ligang Liu, Jiansong Deng, and Falai Chen. Decoupling noise and features via weighted ℓ_1-analysis compressed sensing. *ACM Trans. Graph.*, 33(2):18:1–18:12, 2014. doi: 10.1145/2557449.

[125] Eric W. Weisstein. Triangle area, 2015. URL `http://mathworld.wolfram.com/TriangleArea.html` (last checked 2015-06-17).

[126] Eric W. Weisstein. Circumradius, 2015. URL `http://mathworld.wolfram.com/Circumradius.html` (last checked 2015-06-17).

[127] Eric W. Weisstein. Inradius, 2015. URL `http://mathworld.wolfram.com/Inradius.html` (last checked 2015-06-17).

[128] Eric W. Weisstein. Trilinear coordinates, 2015. URL `http://mathworld.wolfram.com/TrilinearCoordinates.html` (last checked 2015-06-17).

[129] William Allen Whitworth. *Trilinear coordinates and other methods of modern analytical geometry of two dimensions: an elementary treatise*. Deighton, Cambridge, 1866.

[130] Changchang Wu. Towards linear-time incremental structure from motion. In *Proc. Int. Conf. 3D Vision*, pp. 127–134, 2013. doi: 10.1109/3DV.2013.25.

[131] Yong Wu, Yuanjun He, and Hongming Cai. Qem-based mesh simplification with global geometry features preserved. In *Proc. 2nd Int. Conf. Comput. Graph. and interactive techniques in Australasia and South East Asia*, pp. 50–57, 2004. doi: 10.1145/988834.988843.

[132] Julie C. Xia and Amitabh Varshney. Dynamic view-dependent simplification for polygonal models. In *Proc. 7th Conf. Visualization '96*, pp. 327–ff., 1996. doi: 10.1109/VISUAL.1996.568126.

[133] Shiyao Xiong, Juyong Zhang, Jianmin Zheng, Jianfei Cai, and Ligang Liu. Robust surface reconstruction via dictionary learning. *ACM Trans. Graph.*, 33(6):201:1–201:12, 2014. doi: 10.1145/2661229.2661263.

[134] Yizhou Yu. Surface reconstruction from unorganized points using self-organizing neural networks. In *Proc. IEEE Conf. Visualization 99, Late Breaking Hot Topics*, pp. 61–64, 1999.

[135] Gabriel Zachmann and Elmar Langetepe. Geometric data structures for computer graphics. In *Proc. of ACM SIGGRAPH – Tutorial*. 2003. URL `http://www.gabrielzachmann.org/` (last checked 2015-06-17).