Christoph Schinko, Dipl.-Ing.

Shape Processing for Content Generation

DOCTORAL THESIS

to achieve the university degree of

Doktor der technischen Wissenschaften

submitted to

Graz University of Technology

Supervisor

Dieter W. Fellner, Univ.-Prof. Dipl.-Ing. Dr.techn.

Institute of Computer Graphics and Knowledge Visualization

Graz, October 2017

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present doctoral thesis.

Date

Signature

Kurzfassung

In seiner Dissertation „Shape Processing for Content Generation“ behandelt Christoph Schinko generative Modellierung, neuartige Anwendungen der inversen generativen Modellierung sowie Visualisierungssysteme. Diese Bereiche werden als Bestandteile der Formverarbeitung betrachtet, daher richtet sich der Aufbau der Dissertation danach.

Nach der Definition des Begriffs „Form“ befasst sich der erste Teil der Dissertation mit verschiedenen Möglichkeiten der Formbeschreibung. Während einige Formbeschreibungen von abstrakter Natur sind, können andere direkt verwendet werden – beispielsweise auf dem Gebiet der computergestützten geometrischen Gestaltung. Das Thema „Formmodellierung“ (kurz: Modellierung) ist breit gefächert und umfasst die Modellierung mit Primitiven unter Zuhilfenahme von 3D-Modellierungssoftware oder Szenenbeschreibungssprachen, semantische Modellierung mit Metadaten sowie generative Modellierung mit domänenspezifischen Informationen.

Am Beispiel von Trauringen wird mit Hilfe der Generative Modeling Language (GML), einer domänenspezifischen Sprache für die generative Modellierung, ein Design einer ganzen Produktfamilie erstellt. Bei der Auslieferung des Designs über das Web sind unterschiedlichste Plattformen beteiligt. Dieser Umstand lieferte die Idee zu einem innovativen Metamodellier-Ansatz namens „Euclides“. Das innovative Konzept kombiniert die Unterstützung verschiedener Zielplattformen mit einer anfängerfreundlichen Syntax. Damit wird die Grundlage für die plattformunabhängige Generierung von generativen Bausteinen geschaffen. Dieser Ansatz reduziert den Aufwand für die Implementierung und Pflege generativer Beschreibungen für verschiedene Plattformen erheblich.

Aufbauend auf Arbeiten der inversen generativen Modellierung wird die Analyse von digitalisierten Objekten hinsichtlich Veränderungen und Abnutzung möglich. Das vorgestellte System kombiniert generative Beschreibungen mit rekonstruierten Objekten und führt einen Soll-Ist-Wert-Vergleich durch. Durch Anwendung auf einen anderen Parametersatz der generativen Beschreibung können somit neue Formen erzeugt werden. Mit diesem neuartigen Ansatz ist die Gestaltung von Formen unter gleichzeitiger Verwendung hochfrequenter Details sowie High-Level-Form-Parametern möglich.

Der letzte Schritt im Rahmen der Formverarbeitung befasst sich mit Visualisierungssystemen zur Wahrnehmung und Interaktion mit Formen. In diesem Zusammenhang wird eine neuartige Methode zur Projektion eines kohärenten, nahtlosen und perspektivisch korrigierten Bildes von einem bestimmten Gesichtspunkt aus vorgestellt. Der Ansatz zeichnet sich vor allem durch seine Effizienz aus. Der letzte Beitrag zu diesem Thema beschreibt eine optimierte, autostereoskopische Visualisierung auf Basis von Parallaxbarrieren.

Abstract

The thesis "Shape Processing for Content Generation" by Christoph Schinko presents work on generative modeling, novel applications for inverse generative modeling, and visualization systems. These areas are regarded as steps in the context of shape processing, hence the thesis is structured that way.

After defining the term shape, the first part of the thesis is concerned with shape descriptions. While some shape descriptions are of abstract nature, others can be directly used, for example, in the field of computer aided geometric design. The process of working with shape descriptions is called shape modeling. This topic includes primitive modeling using 3D modeling software or scene description languages, semantic modeling dealing with meta data, and generative modeling using domain specific information.

An application for generative modeling in the context of wedding rings is implemented using a domain specific language for generative modeling – the Generative Modeling Language (GML). The multitude of involved platforms (the GML is implemented in C++, the postfix notation of the language itself is similar to Adobe Postscript, the application is targeted for the web) has inspired the idea to create an innovative meta-modeler approach called "Euclides". Its innovative concept of using a beginner-friendly syntax in combination with translation back-ends for various different platforms presents a foundation for the platform-independent creation of generative building blocks. This approach significantly reduces the effort for implementing and maintaining generative description for different platforms.

Building up on previous work on finding the best generative description of one or several given instances of an object class, an application to analyze digitized objects in terms of changes and damages is presented. The system automatically combines generative descriptions with reconstructed objects and performs a nominal/actual value comparison. By applying the variances of the reconstructed objects to a different parameter set of the generative description, new shapes can be created. With this novel approach, the design of shapes using both low-level details and high-level shape parameters is possible.

The last step in the context of shape processing is concerned with visualization systems for humans to perceive and interact with shapes. In this context, a novel method to project a coherent, seamless and perspectively corrected image from one particular viewpoint using an arbitrary number of projectors is presented. The approach distinguishes itself by being quick and efficient. The last contribution to this topic is describing an optimized stereoscopic display based on parallax barriers for a driving simulator.

Acknowledgments

First of all, I would like to use the opportunity and express thanks to my supervisor Prof. Dr. Dieter W. Fellner and my colleagues at Fraunhofer Austria and the Institute of Computer Graphics and Knowledge Visualization at Graz University of Technology. Especially, I would like to thank Torsten Ullrich for providing valuable feedback during many discussions around the contents of this thesis. His ideas had a significant impact on the direction of my work. Furthermore, I would like to thank all the coauthors of the scientific articles published in the context of our work.

Finishing this thesis would not have been possible without the support, encouragement and quiet patience of my beloved wife Annemarie. She often had to endure my absence, especially during her pregnancy.

This is for Annemarie and our daughter Sophia.

TTD!

Publications

Parts of this thesis have been created with the support of the Austrian Research Promotion Agency, the Forschungsförderungsgesellschaft (FFG), in the context of the research project AEDA (K-Projekt "Advanced Engineering Design Automation").

The work in this thesis has been published in the following articles and conference contributions:

[SVP+17] Christoph Schinko, Thomas Vosgien, Thorsten Prante, Tobias Schreck, and Torsten Ullrich. Search and Retrieval in CAD Databases – A User-Centric State-of-the-Art Overview. *Proceedings of the 12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (GRAPP)*, 12:306–313, 2017.

[KSU16] Ulrich Krispel, Christoph Schinko, and Torsten Ullrich. A Survey of Algorithmic Shapes. *Remote Sensed Data and Processing Methodologies for 3D Virtual Reconstruction and Visualization of Complex Architectures*, 219:498–529, 2016.

[SPH+16] Christoph Schinko, Markus Peer, Daniel Hammer, Matthias Pirstinger, Cornelia Lex, Ioana Koglbauer, Arno Eichberger, Jürgen Holzinger, Eva Eggeling, Dieter W. Fellner, and Torsten Ullrich. Building a Driving Simulator with Parallax Barrier Displays. *Proceedings of the 11th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (GRAPP)*, 11:283–291 2016.

[KEL+15c] Ioana Koglbauer, Arno Eichberger, Cornelia Lex, Jürgen Holzinger, Christoph Schinko, and Torsten Ullrich. Evaluation of driving maneuvers in reality and in an autostereoscopic 3D simulation with integrated eye-tracking. *Proceedings of the Human Factors and Ergonomics Society Europe Chapter 2015 Annual Conference*, 2015.

[KSU15] Ulrich Krispel, Christoph Schinko, and Torsten Ullrich. A Survey of Algorithmic Shapes. *Remote Sensing*, 7:12763–12792, 2015.

[KSH+15] Hyosun Kim, Christoph Schinko, Sven Havemann, Ivan Redi, Andrea Redi, and Dieter W. Fellner. Tiled Projection Onto Deforming Screens. *Computer Graphics and Visual Computing (CGVC)*, 1:35–42 2015.

[KEL+15a] Ioana Koglbauer, Arno Eichberger, Cornelia Lex, Norbert Bliem, Anton Sternat, Jürgen Holzinger, Christoph Schinko, and Mario Battel. Bewertung von Fahrerassistenzsystemen von nicht professionellen Fahrerinnen und Fahrern im Realversuch. *Humanwissenschaftliche Beitrge zur Verkehrssicherheit und Ökologie des Verkehrs, mehr sicheres Verhalten im Strassenverkehr*, 5:86–102, 2015.

[SKU15] Christoph Schinko, Ulrich Krispel, and Torsten Ullrich. Know the Rules - Tutorial on Procedural Modeling. *Proceedings of the 10th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (GRAPP Tutorial Notes)*, 10:27ff, 2015.

[KEL+15b] Ioana Koglbauer, Arno Eichberger, Cornelia Lex, Jürgen Holzinger, Christoph Schinko, and Torsten Ullrich. A Model for Subjective Evaluation of Automated Vehicle Control. *Proceedings of the International Symposium on Aviation Psychology*, 18:PW12, 2015.

[SKUF15] Christoph Schinko, Ulrich Krispel, Torsten Ullrich, and Dieter W. Fellner. Built by Algorithms – State of the Art Report on Procedural Modeling. *Proceedings of the 6th International Workshop on 3D Virtual Reconstruction and Visualization of Complex Architectures (3D-ARCH)*, 6:469–479, 2015.

[SSSS14] Helmut Schrom-Feiertag, Christoph Schinko, Volker Settgast, and Stefan Seer. Evaluation of Guidance Systems in Public Infrastructures Using Eye Tracking in an Immersive Virtual Environment. *Proceedings of the 2nd International Workshop on Eye Tracking for Spatial Research co-located with the Eighth International Conference on Geographic Information Science*, 2:62–66, 2014.

[SBEF14] Christoph Schinko, René Berndt, Eva Eggeling, and Dieter W. Fellner. A Scalable Rendering Framework for Generative 3D Content. *Proceedings of the 19th International ACM Conference on 3D Web Technologies*, 19:81–87, 2014.

[SUF14] Christoph Schinko, Torsten Ullrich, and Dieter W. Fellner. Modeling with High-Level Descriptions and Low-Level Details. *Proceedings of the 8th International Conference on Computer Graphics, Visualization, Computer Vision and Image Processing*, 8:328–332, 2014.

[KSU14] Ulrich Krispel, Christoph Schinko, and Torsten Ullrich. The Rules Behind – Tutorial on Generative Modeling. *Proceedings of the 12th Symposium on Geometry Processing / Graduate School*, 12:2:1–2:49 2014.

[USSF13] Torsten Ullrich, Christoph Schinko, Thomas Schiffer, and Dieter W. Fellner. Procedural Descriptions for Analyzing Digitized Artifacts. *Applied Geomatics*, 5:185–192, 2013.

[US13] Torsten Ullrich and Christoph Schinko. Bibliotheksdienste und semantische Auszeichnungen für digitale Artefakte. *Kulturelles Erbe in der Cloud – Fachtagung "Digitale Bibliotheken"*, 4:68ff, 2013.

[KSHF13] Hyosun Kim, Christoph Schinko, Sven Havemann, and Dieter W. Fellner. Tiled Projection onto Bent Screens using Multi-Projectors. *Proceedings of the 7th IADIS International Conference on Computer Graphics, Visualization, Computer Vision and Image Processing*, 7:67–74, 2013.

[SUF12] Christoph Schinko, Torsten Ullrich, and Dieter W. Fellner. Minimally Invasive Interpreter Construction – How to reuse a compiler to build an interpreter. *Proceedings of the 3rd International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking (Computation Tools)*, 3:38–44, 2012.

[BSK+12] René Berndt, Christoph Schinko, Ulrich Krispel, Volker Settgast, Sven Havemann, Eva Eggeling, and Dieter W. Fellner. Ring's Anatomy – Parametric Design of Wedding Rings. In *Proceedings of the 4th International Conference on Creative Content Technologies*, 4:72–78, 2012.

[SSUF11b] Christoph Schinko, Martin Strobl, Torsten Ullrich, and Dieter W. Fellner. Scripting Technology for Generative Modeling. *International Journal On Advances in Software*, 4:308–326, 2011.

[SUF11] Christoph Schinko, Torsten Ullrich, and Dieter W. Fellner. Simple and Efficient Normal Encoding with Error Bounds. *Proceedings of Theory and Practice of Computer Graphics*, 29:63–66, 2011.

[SSUF11a] Thomas Schiffer, Christoph Schinko, Torsten Ullrich, and Dieter W. Fellner. Real-World Geometry and Generative Knowledge. *The European Research Consortium for Informatics and Mathematics (ERCIM) News*, 86:15–16, 2011.

[SUSF11] Christoph Schinko, Torsten Ullrich, Thomas Schiffer, and Dieter W. Fellner. Variance Analysis and Comparison in Computer-Aided Design. *Proceedings of the 4th International Workshop on 3D Virtual Reconstruction and Visualization of Complex Architectures (3D-ARCH)*, 4:21–25, 2011.

[SSUF10b] Martin Strobl, Christoph Schinko, Torsten Ullrich, and Dieter W. Fellner. Euclides – A JavaScript to PostScript Translator. *Proceedings of the 1st International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking (Computation Tools)*, 1:14–21, 2010.

[SSUF10a] Christoph Schinko, Martin Strobl, Torsten Ullrich, and Dieter W. Fellner. Modeling Procedural Knowledge – A Generative Modeler for Cultural Heritage. *Proceedings of the 3rd International Euro-Mediterranean Conference, (EuroMed)*, 6436:153–165, 2010.

[USF10] Torsten Ullrich, Christoph Schinko, and Dieter W. Fellner. Procedural Modeling in Theory and Practice. *Poster Proceedings of the 18th WSCG International Conference on Computer Graphics, Visualization and Computer Vision*, 18:5–8, 2010.

Contents

Chapter 1

Introduction

Digitally represented 3D content is used in a wide variety of fields. In the movie industry, digital characters and objects are used for the creation of motion pictures. 3D content is used as assets for games by the video game industry. There are also numerous applications in the architectural and engineering community to visualize buildings or construct components using 3D printers or CNC machines.

The many applications for 3D content underline the importance of the need to answer questions in the context of its processing. After an overview of the involved steps, the following chapter introduces the idea behind generative modeling and its connection to the current trend towards product mass customization. Together with a brief introduction to the topic of visualization and virtual reality, these two aspects of 3D content processing form the thematic framework for the contributions presented within this thesis.

Contents

1.1 Digital Technology

Digital technology has changed modern society in many different ways. It has entered into every aspect of our daily lives. Smartphones are a good example to illustrate this development. They can be used to communicate with other people, access the internet, take photographs, track our daily fitness activities, navigate us from one place to another, play games, watch videos, just to name a few tasks. While we used to have different ways to process some of these tasks, many of them only emerged because of the new possibilities. As a side effect, not only smartphones, but all kinds of different devices, sensors and activities create data. Since a lot of our activities are directly related to – if not depending on – the internet, we have created a digital footprint of our lives online.

The term data is closely related to the terms information and knowledge. While data is just a set of values, it becomes information by interpretation. Knowledge, on the other hand, can be seen as experience derived from dealing with information on a specific subject. The data created by (or through) our activities can be associated with a context, and without focusing on aspects like privacy, security, or ownership, offers an immense value, when processed. Our online browsing behavior can be used to identify users, for advertising purposes, to improve the quality of search results, or even by intelligence agencies to identify potential security threats.

A lot of this data is created by different kinds of sensors in our digital devices. The desire to preserve special moments in live is being addressed by sensors trying to capture reality. Sensors capturing 2D information like images or videos have been used before the advent of digital technology and have not lost their importance. On the contrary, they are more and more used (and also accompanied by new sensors) to make devices aware of their surroundings. Tango[1] from Google is an augmented reality computing platform using computer vision to enable new user experiences for mobile devices. Apart from augmented reality, environmental recognition or physical space measurements, this technology can be used to create a 3D representation of our surroundings (also called 3D reconstruction).

But the devices are not only capable of 3D reconstructing our surroundings, they are also powerful enough to create immersive virtual experiences. Samsung Gear VR[2] is a mobile virtual reality headset using smartphones as display and processing unit. Some head-mounted devices, like Microsoft HoloLens[3], even combine both, awareness of our surroundings and virtual reality, to create mixed reality experiences. Many applications will benefit from virtual and mixed reality, like indoor and outdoor navigation, support for maintenance tasks, or product configurators.

1.1.1 3D Content Processing

Especially the developments in the fields of virtual and mixed reality emphasize the importance of 3D content and how to answer questions in the context of its processing. This task resembles a pipeline and is concerned with representation, modeling, delivery and visualization.

[1] https://developers.google.com/tango/
[2] https://www.samsung.com/global/galaxy/gear-vr/
[3] http://hololens.com/

Depending on context and use, there are different ways to represent 3D content. However, the choice of representation has a direct effect on possible modeling paradigms. Modeling operations heavily depend on their underlying representation. While conversions between many representations are possible (and necessary), they do not always yield satisfying results. Thus, many problems can be avoided by choosing a suitable representation first.

After modeling, 3D content has to be prepared for delivery to and visualization on the target platform. Encoding and decoding of 3D content needs to be adapted to possible limitations of the platform, like available bandwidth and hardware resources. The size of a model – both, spatially and in file size – plays an important role regardless of the platform. Typically, a trade-off between file size and computational effort has to be found, keeping in mind that graphics hardware expects 3D content to be available in the form of triangles.

At this point, it is important to emphasize, that this processing pipeline (henceforth called shape processing pipeline) establishes a thematic framework for all contributions presented in this thesis – it does not claim to be comprehensive.

1.2 Generative Modeling

While describing 3D content on the geometric level is a problem that has been researched reasonably well, it is still an open question how to describe its structure on a higher, more abstract level [LZQ06]. In that sense, generative modeling offers a possibility to represent 3D content with a sequence of generating operations, and not just with a list of low-level geometric primitives, like triangles or surface patches. Large scale models and scenes can therefore be represented and delivered efficiently.

Generative modeling resembles encoding information in some form of programming language. Depending on the specific task, special purpose languages and tools are used. Such languages and tools are good at solving narrowly defined, domain-specific problems (e.g., procedural architecture [LWW08], or the creation of natural branching structures [LD98]). However, their embedding into larger systems can be challenging due to dependencies to scripting and rendering engines.

A major benefit of generative modeling techniques is, that they make complex 3D content manageable by allowing to identify its inherent high-level parameters. This property is especially important in the context of product mass customization.

1.2.1 Mass Customization of Products

The current trend towards computerization in manufacturing originates from a project of the German government called *Industrie 4.0* [Bun17]. A main focus is to create smart factories consisting of cyber-physical systems, the internet of things and cloud computing. The idea behind cyber-physical systems is to virtually represent physical processes to allow automatic controlling and monitoring. Creating virtual representations has only become possible by fitting small computers and sensors into objects, thus creating the internet of things. The potential mass of data created by an ever growing number of smart devices

can be coped with cloud computing approaches. Not only do they provide the power and flexibility to process data, but they also allow internal and external services to be offered in the context of the value chain.

At the same time, a trend towards mass customization of products can be observed, for example, polyethylene terephthalate (PET) bottles with individual motifs, or cars that can be configured to the level of choosing color and material of buttons on the dashboard. While certain products, like wedding rings, have always been very individual, factories designed for mass production of products face a problem. So far, batch sizes of at least a few thousand identical products still allow for manual scheduling and intervention. With the trend towards batch size one scenarios in production, this is no longer feasible. Smart factories are becoming a key element for this development.

While manufacturers are adapting to this trend, the possibilities for customers to configure or design products also play an important role. Since a huge amount of configuration options cannot be communicated satisfactorily in a brochure, other means of representation have to be found. The advent of native 3D support for web browsers laid the technical foundation for the creation of online 3D product configurators.

A generative description of a product offers the advantage to not only cover possible configuration options, but also allows to change its shape by altering high-level parameters. Thus, it is possible to steer the customization process into meaningful directions, while still offering creative freedom. In contrast to more primitive shape manipulations, this paradigm enables the designer to ensure important properties like manufacturability or meaningfulness. A simple example illustrates these properties: a configurator for channeled sheet. The configurator allows the sheet to be resized to individual extent. For this purpose, the resize operation has to take into account, that the pattern of the sheet needs to retain its size. A simple scaling operation is not sufficient – the resize operation needs to be more “intelligent”. These aspects can be covered with generative modeling techniques.

The generative encoding of 3D content enables the representation of whole product families. Single instances are evaluated with a specific set of parameters. This flexibility comes at the cost of more computational effort for visualization.

1.3 Visualization

Visualization, in general, is concerned with creating images to transport abstract and concrete ideas. It has been used to present information in the form of drawings and maps for over thousand years. An early example of a visualization is the map by Charles Joseph Minard (see Figure 1.1).

The map is representing the successive losses in men of the French army in their Russian campaign (1812-1813). The illustration depicts the size of the army departing at the Polish-Russian border up until their retreat. Additional information about distance and direction traveled, temperatures, as well as location is incorporated into the map. Due to the fact that the map incorporates a lot of data while still being easily comprehensible, it is regarded as a good example of information visualization.

Apart from the invention of the central perspective in the Renaissance period, the advent of computer graphics marked an important development in the

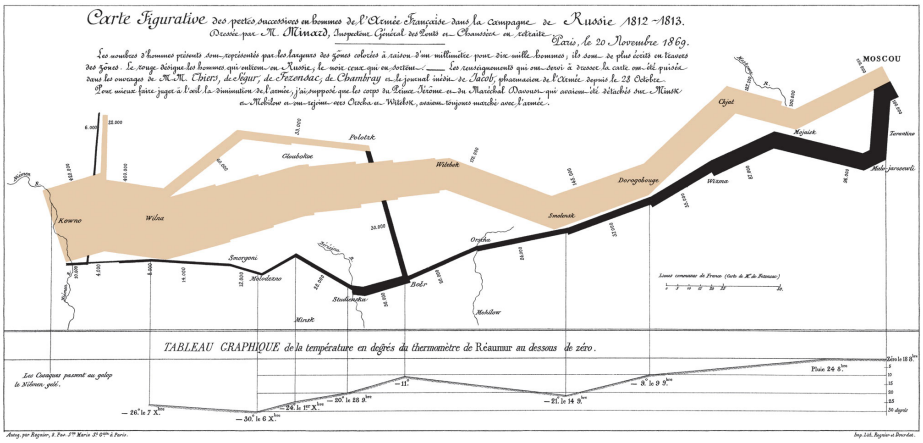


Figure 1.1: The figurative map of the French Russian campaign (1812-1813) shows the successive losses in men of the French army.
(Source: Charles J. Minard, 1869
http://en.wikipedia.org/wiki/File:Minard.png)

field of visualization. Together with the development of moving images (animation), it was now possible to exploit new ways of digitally producing and interacting with images.

Depending on the application, there are two main paradigms for creating images: the interactive creation in real-time (i.e., within a tight timeframe), and the creation of animations. The difference is that a user can control an interactive visualization, whereas an animation is a sequence of already produced images (or just a single image). The tight timeframe for interactive visualizations limits the degree of realism compared to animations. This is due to the fact that all images need to be created (rendered) within this timeframe.
Generally, two techniques for creating images are used in computer graphics, although a distinction between ray casting and ray tracing can be made (ray casting algorithms never spawn secondary rays [Wei07]):

- **Rasterization** is the task of converting 3D content (or, more general, vector information) into (raster) images by geometrical projection onto an image plane.
- **Ray tracing** is a technique to create an image by tracing the path (regarded as a ray) of light through pixels in an image plane while taking into account the physical interaction with the 3D content.

Ray tracing is capable of producing a very high degree of realism, but at the cost of high computational effort, making it best suited for animation purposes. For interactive visualizations, rasterization techniques offer the necessary performance. The creation of images, whether it be interactive or not, is the prerequisite for presenting them to the user. This is typically done using displays or projection systems. Depending on the area of application, these techniques can be used to create a broad range of experiences using simple computer displays, or complex virtual reality installations.

DAVE

In an effort to reduce the costs of cave automatic virtual environments (CAVE) installations, the Institute of Computer Graphics and Knowledge Visualization (CGV) at the University of Technology in Graz, Austria built the DAVE. DAVE stands for definitely affordable virtual environment. It achieves its goal of being affordable by mostly using standard hardware components compared to other systems. Marcel Lancelle and Volker Settgast thoroughly describe various aspects of the DAVE in their PhD theses [Lan11] [Set13]. The DAVE itself resembles the shape of a box with 3.30m wide walls consisting of three back-projected screens (left, right, front), and one front-projected screen at the bottom (see Figure 1.2).

Large mirrors are used to fold the light paths from the projectors to the screens to minimize the required room size. Eight computers are connected to four projectors responsible for the active stereo projection. An optical system with four cameras is responsible for tracking a single user inside the DAVE. Due to the fact that standard hardware is used for computers and graphics cards, the only relatively expensive parts of the DAVE are its projectors.

Figure 1.2: The DAVE in Graz resembles the shape of a box with 3.30m wide walls and consists of four projection walls (left, right, front, bottom) and four active stereo projectors being driven by eight computers.

1.3.1 Virtual Reality

The term virtual reality stands for a technology used to digitally replicate a real environment for a user to interact with. Apart from realistic renderings, specialized screens, projectors, devices like headsets, or even large CAVE installations are used to create immersive, interactive experiences. Jason Jerald [Jer16] defines virtual reality as " [...] a computer-generated digital environment that can be experienced and interacted with as if that environment were real." However, virtual reality is not limited to visual experiences, but can include audio, haptic, and, potentially also, olfactory feedback. A general distinction can be made between (more or less mobile) head-mounted devices and static systems using displays or projectors surrounding the user. For head-mounted devices, correct rotation of the virtual camera is ensured using motion sensors, or an external tracking system. The user can experience a full 360 degree view or even walk around in the virtual world. Head-mounted device are cheaper, smaller, and mobile in contrast to static systems, like CAVE installations, where stereoscopic projection and head tracking are used to display the correct perspective per eye. CAVE systems are multi-user friendly and generally offer a more immersive experience due to the fact that users can see themselves.

For a long time, virtual reality used to only be possible through CAVE installations and was therefore more of a niche product due to high costs and space requirements. As mentioned earlier, this changed rapidly with the advent of mobile virtual reality headsets. In general, advances in displays and projection systems are the driving force for new visualization techniques tailored towards specific use cases.

1.4 Open Problems and Contributions

There are still many open questions in the context of the shape processing pipeline. Choosing a proper representation for 3D content heavily depends on the use case. However, the majority of shape representations are tailored towards describing a shape at the geometric level and do not incorporate its inherent structure. This is especially important in the context of manufacturing and product configuration, where an effective representation of a design space of a product together with its numerous configuration options can be of great benefit. While generative aspects of a product are starting to be exploited by manufacturers during the construction process, they need to also be consequently implemented for customers using configurators. Different tools and platforms are typically used at these two stages of the product cycle to represent similar (if not the same) aspects. The effort to implement and maintain several generative descriptions is significant.

To address this problem, a novel meta-modeling approach called *Euclides* is presented within this thesis. It addresses the problem of implementing and maintaining generative descriptions on different platforms. From a single implementation of all necessary construction rules, Euclides generates executable code for a variety of different target platforms. Its high-level representation of the input code allows to preserve the level of abstraction when translating to a target platform. The system creates target code with a clear correspondence to the input code, thus simplifying debugging and reuse. Additionally, its easy-to-use language based on JavaScript helps lowering the inhibition threshold for non-computer experts to use this system.

When generative descriptions are used to resemble real-world objects, differences between the generative description and the real-world object can be a result of a lack of detail in the generative description. These deficiencies are addressed within this thesis by introducing a method to offset possible fine details of a real-world object onto the generative description. This technique can be used to design shapes using both low-level details and high-level shape parameters at the same time. The additional template transfer step enables the generation of new shapes by altering the parameters of the generative description.

The shape processing pipeline is also concerned with proper visualization of 3D content tailored towards specific use cases. An abstract visualization of artistic nature has different requirements than a realistic visualization for a driving simulator. This not only has an impact on the choice of shape representation, but also on the visualization hardware. Covering the artistic aspect of visualization (not necessarily limited to 3D content), a system to seamlessly project onto non-planar surfaces is presented within this thesis. It is used to create dynamic, reconfigurable spaces influencing the behavior of groups and in-

dividuals. In contrast, this thesis also presents an immersive, autostereoscopic visualization system for a driving simulator. Both systems heavily differ in their requirements showing the diverse application fields for visualization.

1.5 Overview

This thesis is structured following the shape processing pipeline. It is divided into seven main parts. The next chapter is focused on shape representations. It is followed by a chapter on different modeling paradigms and delivery of generative content. The meta-modeler framework Euclides is presented in the fourth chapter. Inverse modeling and techniques for modeling with low-level details and high-level shape parameters are the main topic of the fifth chapter. The sixth chapter presents a system for non-planar, tiled projections as well as an autostereoscopic visualization system for a driving simulator. The last chapter is concerned with future work and concludes this thesis.

Furthermore, grey boxes (so-called *Notes*) can be found throughout the thesis. The rationale behind using these special areas is to give additional information related to the content they are embedded in. Incorporating these areas into the text would have required additional sections, or even chapters resulting in an unnecessary bloating of the thesis.

Chapter 2

Shapes

After giving a general definition of the term shape, this chapter focuses on providing a structured introduction to different ways of describing and representing shapes. The intent behind this chapter is to point out differences between lower-level descriptions relying on more geometric primitives, and higher-level descriptions being of more abstract nature.

Starting with two brief sections about textual and image-based descriptions, the following section gives a more detailed overview of point sets, polygonal faces, as well as parametric and implicit surface representations. The section about volumetric shape descriptions is concerned with voxels, convex polytopes and constructive solid geometry. Finally, algorithmic shape descriptions are introduced and characterized.

The following sections do not claim to be an exhaustive collection of possible shape descriptions, but a sufficiently complete set for all further considerations.

Contents

2.1 Definition and Perception

"A shape is a limited arrangement of straight lines defined in a Cartesian coordinate system with real axes and an associated Euclidean metric." This definition of shape has been used by George Stiny in his work on shape and shape grammars [Sti80]. While this definition may be sufficient enough for working with two-dimensional shapes and shape grammars, it lacks a generic nature to be applied to more complex shapes in higher dimensions. In this thesis the term shape is defined by a more general formulation, thus including a larger selection of geometric primitives like Bézier curves, triangles, subdivision surfaces:

Definition A shape is the particular physical form or appearance of an object.

This definition is following the more general definition of shape from:
http://dictionary.cambridge.org/

Many different ways of shape descriptions are available, tailored to the requirements in their respective areas of research. In the context of Computer-Aided Design (CAD), the model description of a digital counterpart of a real object is called a shape description.

At this point, it is important to emphasize that there are differences in the process of shape perception between human beings and computers. On the one hand, there are sensory differences. In their natural surrounding, human beings can rely on their five senses to perceive a shape. Consequently, it is often a combination of these senses that makes up the sensation of a shape. While computers can be fitted with many different sensors, adding up to far more different senses compared to human beings, it usually boils down to a specific sensor being used to perceive a shape. The reason for that circumstance is directly related to the second aspect in this context – the reasoning itself. The human brain is yet to find a matching rival in the world of computer science. While computers are programmed to outperform the human brain in various, but rather specific tasks like number crunching, the computer is no thinking machine. Interdisciplinary developments in all fields of computer science over the recent years bring us ever closer to creating the thinking machine. However, especially for a computer, the task of shape classification heavily depends on the underlying description. Even after successfully classifying shapes, a computer is yet not aware of the meaning of shape, as discussed by Sven Havemann, Torsten Ullrich and Dieter W. Fellner in their work [HUF12]. For the description of shape, it is important to be aware of these differences, even if shape classification is not in the context of this thesis.

2.2 Textual Shape Descriptions

In dictionaries, shapes are described by words forming a textual definition:

ring a typically circular band of metal or other durable material, especially one of gold or other precious metal, often set with gems, for wearing on the finger as an ornament, a token of betrothal or marriage, etc.

http://dictionary.reference.com

Search & Retrieval in CAD Databases

It is an algorithmic challenge to perform search and retrieval tasks on different shape respresentations. The various approaches to tackle this problem stem from different research domains (e.g., geometry processing, computer vision, pattern matching, knowledge management). In the article "Search & Retrieval in CAD Databases – A User-Centric State-of-the-Art Overview" [SVP+17] by Christoph Schinko et al., we present a state-of-the-art overview on shape, information and design retrieval systems in the context of CAD engineering.

Unlike retrieval systems for multimedia databases, 3D shape repositories in an engineering context are facing several challenges [JKIR06]:

1. Engineering shapes are characterized by features such as holes, tunnels, cavities, etc. The relative position of these features are more important for a part's functionality than its overall shape.

2. The classification of parts in the engineering context has a low level of abstraction; e.g. a category "airplanes" is not very reasonable in the context of CAD, as an airplane would be considered as an assembly of many much smaller objects.

3. In the CAD context, parts are often classified according to their functionality and not according to their shape.

The classification of shape and information retrieval systems from a CAD application user's point of view consists of four categories inspecting different aspects of a digital library application:

- **Data and meta data representation** The first category analyzes the design retrieval approaches according to supported input data. This criterion does not address file formats but the geometric entities (point clouds, polygonal meshes, ...) and the non-geometric entities (annotations, material properties, ...) which can be handled natively without conversion.

- **Queries and results** This category is concerned with the types of queries that are supported (by example, by an image, by a sketch, ...), as well as the possibility to perform subpart matching.

- **Technology readiness level** The availability level of the presented approach (only article published, reference implementation available, ...), as well as how an available implementation can be integrated into an existing CAD environment is covered by this category.

- **Technology** The last category is concerned with an analysis of the search method and its performance. A primary search method uses the (geometric and non-geometric) content. A secondary search method relies on a primary search method and uses additional sources not contained in the content.

From a computer science point of view, this definition is of a rather abstract nature representing a difficult basis for creating detectors. A computer program relies on more formal, mathematical definitions.

For a human being this description is sufficient enough to easily recognize the very shape of a ring when seeing it. The precondition for this accomplishment is a basic understanding of the terms and definitions used in the description. Bootstrapping of the basic concepts on a textual basis alone is hardly possible. All available senses are used to create a mental image of the surrounding environment, making it possible to establish a connection between sensory input and concepts of shapes. Then, a single sensory input is enough for the brain to be made aware of the related concepts. As an example, the human visual system is capable of identifying and categorizing objects. Not only real-world objects, but also schematic drawings, pictures, paintings, etc. can serve as input. The description itself can also be available in the form of an image.

2.3 Image-based Shape Descriptions

Representation and description of shapes in images is one of the basic methods to describe image content. However, similar to textual descriptions, image-based descriptions are a rather informal definition of shape, thus representing a difficult basis for creating algorithmic detectors. This is due to the loss of one dimension of object information when projecting an object onto the image plane. Shape information in images is often also affected by noise, distortion and occlusion. As a result, the shape extracted from the image only partially represents the real world object. Figure 2.1 shows a pair of wedding rings.

Figure 2.1: The pair of wedding rings is easily identifiable for a human even though there is not enough information to fully describe the shape of both rings. It is a considerably harder problem for a computer to describe, or even identify the rings.

Due to the limited amount of information in this image it is impossible to fully describe the shape of both rings. For a human, the amount of information is sufficient enough to identify the pair of rings. However, it is a considerably harder problem for a computer to describe, or even identify the rings.

Semantic scene understanding is concerned with finding a connection between pixels of an image and what is considered to be the meaning, or the contents of an image [SEFR14]. Descriptors play an important role in this context. Because of the large number of descriptors for visual features in images, a classification of the descriptors related to shape is sufficient for this thesis. Dengsheng Zhang and Guojun Lu classify these descriptors into two classes of methods: contour-based methods and region-based methods [ZL04]. Shape features extracted from the contour only belong to a different class than shape features extracted from the whole shape region. These methods can be further divided into structural approaches and global approaches reflecting the difference of shapes being represented as a whole or by primitives or sub-parts. A final distinction into space domain and transform domain is based on whether the shape features are derived from the spatial domain or from the transformed domain.

2.4 Surface-Based Shape Descriptions

The surface of a shape is the part that is forming its boundary and thus is visible. In many applications the surface is a sufficient description of a shape, e.g., collision detection. Describing a shape through its surface is also a common practice in computer graphics.

Surface-based shape descriptions are generally more formal than textual shape descriptors. Depending on the specific details of the description, the algorithmic detection of shapes has to rely on identifying features, e.g., salient points, or relies on using skeletal or topological graph structures. A recent survey by Bo Li et al. describes different techniques and compares current approaches [LLL+15]. Proper visualization is important for human beings to perceive a surface-based shape description.

2.4.1 Point Sets

Points are a basic primitive to describe the surface of a shape [ZPKG02]. A point set is a list of points defined in a coordinate system. While points are not the primitive of choice when using 3D modeling software to create shapes, they are widely used by 3D scanners due to the nature of their measurements. A point set is the outcome when measuring a large number of points on an object's surface. The data set in Figure 2.2 shows the laser scan of a plastic duck consisting of 177 264 points. The number of points in this example poses no problem for rendering on modern hardware.

High resolution scans of larger objects require special techniques due to the huge amount of data. For different rendering approaches the literature survey of Markus Gross and Hanspeter Pfister offers in-depth explanation [GP07]. Point sets are seldom directly usable in 3D applications. The creation of a shape from point set data is called shape reconstruction.

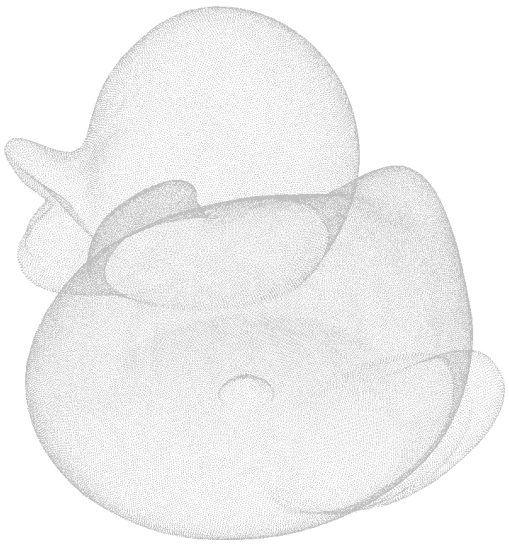


Figure 2.2: The laser scan of a plastic duck consisting of 177 264 points poses no problem for rendering on modern hardware. Additional details like texture or normal information would greatly increase the expressiveness of the visualization.

2.4.2 Polygonal Faces

A very common representation to describe a shape's surface is to use a mesh of polygonal faces. The accuracy of the representation heavily depends on the shape's outline and is directly affected by the number of faces. A cylinder, for example, cannot be accurately represented by planar faces – it can only be approximated, see Figure 2.3. Curved surfaces, in general, cannot be represented exactly, whereas objects having planar boundaries obviously can be.

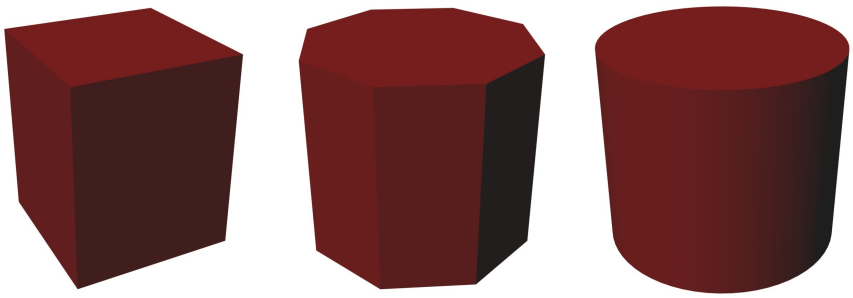


Figure 2.3: The quality of the polygonal approximation of a cylinder highly depends on the number of primitives. By increasing the number of primitives a better approximation of a cylinder can be created. These examples have n-gonal prisms, where n is 4 (left), 8 (middle), 64 (right).

This limitation is often outweighed by its advantages in the field of CAD:

- Computer graphics hardware is tailored towards processing polygonal faces – especially triangles. This is the reason why many of the other shape representations are converted into polygonal meshes prior to rendering.
- A lot of tools and algorithms exist to create, process and display polygonal objects [BK10], [AGFF09].

The data structures for storing polygonal meshes are manifold. In a very simple form, a list of coordinates $(x,\ y,\ z)$ representing the vertices of the polygons can be used. The de-facto standard data interface between CAD software and machines (e.g., milling machines, 3D printers, etc.), which is the STereoLithography (STL) file format, is based on this data structure. In an STL file, the simplest polygon, namely the triangle, is used.

While this data structure is sufficient for manufacturing purposes, it may not satisfy the needs of a 3D modeler in terms of attribution, traversal and editing. More sophisticated data structures reproducing hierarchical structures (groups, edges, vertices) and adding additional attributes like normals, colors and texture coordinates provide a remedy. The problem of traversing a mesh can be tackled by introducing iterators. They are typically, but not exclusively, used in combination with the concept of half-edges. The idea is to represent an edge between two vertices by two half-edges of opposite direction. A half-edge is a directed edge with references to its opposite half-edge, its incident face, vertex and next half-edge. By defining operations using this data structure, it is possible to conveniently traverse a mesh [BSBK02].

2.4.3 Parametric Surface Representations

A parametric representation of a shape's surface is defined by a vector-valued parameterization function $f : \Omega \to S$ mapping a 2D parameter domain $\Omega \subset \mathbb{R}^2$ to the surface $S = f(\Omega) \subset \mathbb{R}^3$. This representation is a general way to specify a surface. A simple 3D example of a parametric surface is the torus with major radius R and minor radius r defined by the range of the parametric function

$$f : [0, 2\pi] \times [0, 2\pi] \to \mathbb{R}^3, \qquad (u, v) \mapsto \begin{pmatrix} \cos(u)(R + r\cos(v)) \\ \sin(u)(R + r\cos(v)) \\ r\sin(v) \end{pmatrix}.$$

The following theorem by Karl Weierstrass states, that finding an explicit formulation with a single function approximating a more complex function (shape) can be achieved by using polynomials.

Theorem (Weierstrass Approximation Theorem)**.** *Let f be a continuous real-valued function on the closed interval $[a, b]$. Then f can be uniformly approximated by polynomials.*

Parametric and Geometric Continuity

The smoothness of a surface is a very important aspect for many different applications, for example in car design. It can be formalized according to the properties of a surface's derivatives. For parametric continuity, the surface is treated as a function rather than a shape, because it cannot be defined given only the shape of the surface – a parameterization is needed. The various orders of parametric continuity for curves can be described as follows:

- **C^0 continuous** The junction of two curves is C^0 continuous, if the coordinates (the x, y, z values of the two curves) agree – the curves are joined. This is called zero order parametric continuity.
- **C^1 continuous** The junction of two curves is C^1 contin-uous, if the coordinates (the x, y, z values of the two curves) agree, and all their first derivatives also agree at their junction – the first derivatives are continuous. This is called first order parametric continuity.
- **C^2 continuous** The junction of two curves is C^2 continuous, if the coordinates (the x, y, z values of the two curves) agree, and all their first and second derivatives agree at their junction – the first and second derivatives are continuous. This is called second order parametric continuity.

The definition of the n^{th}-order parametric continuity C^n requires all derivatives up to n^{th}-order to agree at the junction.

Geometric continuity, on the other hand, can be defined on the shape of the curve alone:

- **G^0 continuous** The junction of two curves is G^0 continuous if the coordinates (the x, y, z values of the two curves) agree. This is called zero order geometric continuity and is exactly the same as C^0 continuity.
- **G^1 continuous** The junction of two curves is G^1 continuous if the coordinates (the x, y, z values of the two curves) agree, and all their first derivatives are proportional at their junction (the tangent vectors are parallel). The curves also share a common tangent direction at the junction. This is called first order geometric continuity.
- **G^2 continuous** The junction of two curves is G^2 continuous if the coordinates (the x, y, z values of the two curves) agree, and all their first and second parametric derivatives are proportional at their junction. The curves also share a common center of curvature at the junction. This is called second order geometric continuity.

The junction of two curves is G^n continuous if their arc length parameterizations meet with C^n continuity.

Continuity measurements have practical use. In the previously mentioned example of car design, G^3 continuity assures smooth reflections in the car body.

A constructive proof of the theorem is given by Sergi Bernstein through his work on Bernstein polynomials (see Section 2.4.3). As a consequence, the concept of surface patches has gained currency in the CAD domain [Far90], [PL02]. The idea is to split the function domain into smaller regions. Each surface patch, henceforth called patch, is described by a distinct parametric function approximating the local geometry of the patch. To obtain a good overall approximation of the surface, it is necessary to carefully chose the layout of the patches (form, size, number) and to deal with possible discontinuities on patch borders depending on the representation [HL89].

Bézier Surfaces

A Bézier surface is a three-dimensional surface generated from the Cartesian product of two Bézier curves [PBP02]. A Bézier surface of degree (m, n) is defined as

$$f(u,v) = \sum_{i=0}^{m} \sum_{j=0}^{n} b_{ij} B_i^m(u) B_j^n(v).$$

It is evaluated over the unit square $(u, v) \in [0, 1] \times [0, 1]$ with the control points b_{ij} and the Bernstein polynomials

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}$$

of degree n, for $t \in [0, 1]$. In CAD, Bézier surfaces are often used in the form of bicubic Bézier patches. In this case, a set of 4×4 points represents the control mesh and is responsible for the shape of the surface of a bicubic Bézier patch. In all cases, Bézier surfaces have important properties:

- A Bézier surface can be seen as a Bézier curve moving along another Bézier curve creating a surface.
- A Bézier surface fulfils the partition of unity property;
  i.e., $\sum_{i=0}^{m} \sum_{j=0}^{n} B_i^m(u) B_j^n(v) = 1$, thus the relationship between a Bézier surface and its control mesh is invariant under affine transformations.
- A Bézier surface is contained within the convex hull of its control mesh and the four corner points of the control mesh are interpolated by the Bézier surface. A point set S is convex, if for any pair of points $x, y \in S$, the line segment $\lambda x + (1 - \lambda)y$ with $0 \leq \lambda \leq 1$, lies entirely in S. For any set S, the smallest convex set containing S is called the convex hull of S.
- A Bézier surface exhibits four boundary curves being Bézier curves themselves and their control points are the boundary points of the control mesh.
- The control points do not exert local control alone. When moving a single control point, the whole surface of the patch is affected, even magnitude and direction of the tangents. Higher order geometric continuity (e.g., G^1, G^2) between patch segments can only be achieved by satisfying constraints when moving boundary control points.

Rational Bézier Surfaces

The idea behind rational Bézier surfaces is to add adjustable weights to extend the design space of shapes [AS93]. In contrast to a Bézier surface, which can only approximate spheres and cylinders, the rational Bézier srfaces can describe them exactly – a very important property in CAD. A rational Bézier surface of degree (m, n) is defined as

$$f(u,v) = \frac{\sum_{i=0}^{m}\sum_{j=0}^{n} w_{ij} b_{ij} B_i^m(u) B_j^n(v)}{\sum_{i=0}^{m}\sum_{j=0}^{n} w_{ij} B_i^m(u) B_j^n(v)}$$

with the control points b_{ij}, the weights w_{ij}, and the Bernstein polynomials $B_i^m(u)$ and $B_j^n(v)$. Rational Bézier surfaces are a special case of non-uniform, rational B-spline (NURBS) surfaces, which are a generalization of B-Spline Surfaces.

B-spline Surfaces

B-spline surfaces exhibit advantages when joining patches under continuity requirements. Let $k, l, m, n \in \mathbb{N}$ with $m \geq k$ and $n \geq l$. Then, a B-spline surface of degree (k, l) is defined as

$$f(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n} d_{ij} N_i^k(u) N_j^l(v),$$

with the basis functions $N_i^0(t) = 1$, if $t_i \leq t < t_{i+1}$ (and 0, otherwise) and

$$N_i^r(t) = \frac{t - t_i}{t_{i+r} - t_i} N_i^{r-1}(t) + \frac{t_{i+1+r} - t}{t_{i+1+r} - t_{i+1}} N_{i+1}^{r-1}(t)$$

for $1 \leq r \leq n$ and a nondecreasing sequence of knots, a so-called knot vector,

$$T = \{t_0 \leq \cdots \leq t_n \leq \cdots \leq t_{n+m+1} t\}.$$

It can be evaluated over $(u, v) \in [u_k, u_{m+1}[\times[v_l, v_{n+1}[$ with the control points d_{ij} and the polynomials $N_i^k(u)$ and $N_j^l(v)$. The control points d_{ij} forming the control polygon are called de Boor points. In computer graphics, B-spline surfaces are typically used in the form of bicubic B-spline patches. A single cubic B-spline curve segment is defined by four control points; as a consequence, 4×4 control points define a bicubic B-spline patch segment. B-splines with knots t_i satisfying the condition $t_0 = 0$ and $t_{i+1} = t_i$ or $t_{i+1} = t_1 + 1$, $(i = 0, \ldots, n + m)$ are called uniform B-splines.

B-spline surfaces satisfy properties similar to Bézier surfaces [PBP02]:

- The relationship between a B-spline surface and its control mesh is invariant under affine transformations.
- A B-spline surface is contained within the convex hull of its control mesh.
- In contrast to Bézier surfaces, the control points exert local control – if a control point is moved, only the local neighborhood is affected.
- By choosing appropriate knot vectors, a B-spline surface can become a Bézier surface.

Higher order geometric continuity (e.g., G^1, G^2) when combining B-spline patches can be achieved by satisfying constraints when moving boundary control points and by appropriate choice of knot vectors. In contrast to Bézier surfaces, moving control points to achieve certain continuity conditions only affects the surface locally.

NURBS Surfaces

The combination of rational Bézier techniques and B-Spline techniques leads to non-uniform, rational B-Splines; NURBS for short [PT97]:

Let $k, l, m, n \in \mathbb{N}$ with $m \geq k$ and $n \geq l$. Additionally, let $w_{00}, \ldots, w_{mn} \in \mathbb{R}$, $\mathrm{u} = (u_0 \ldots u_{m+k+1})^T$ and $\mathrm{v} = (v_0 \ldots v_{n+l+1})^T$ be two knot vectors and $d_{00}, \ldots, d_{mn} \in \mathbb{R}^3$. Then, a non-uniform rational B-spline (NURBS) surface of degree (k, l) is defined as

$$f(u,v) = \frac{\sum_{i=0}^{m} \sum_{j=0}^{n} w_{ij} d_{ij} N_i^k(u) N_j^l(v)}{\sum_{i=0}^{m} \sum_{j=0}^{n} w_{ij} N_i^k(u) N_j^l(v)}$$

over $(u, v) \in [u_k, u_{m+1}[\times[v_l, v_{n+1}[$ with the control points d_{ij}, the polynomials $N_i^k(u)$ and $N_j^l(v)$, the knot vectors u and v for the de Boor points $d_{00}, \ldots, d_{mn}$ and the weights $w_{00}, \ldots, w_{mn}$. Similar to B-spline patches, NURBS surfaces are commonly used in computer graphics in the form of bicubic NURBS patches.

Both B-spline surfaces and Bézier surfaces are special cases of NURBS surfaces [FLS04]. If all weights are equal, a NURBS surface becomes a B-spline surface. Additionally, when all knot vectors are chosen appropriately, the B-spline surface becomes a Bézier surface.

NURBS surfaces are a generalization of B-Spline and Bézier surfaces. Affine transformations are applied to the surface by applying them to the control points. A NURBS surface is generally not contained within the convex hull of its control mesh. Depending on the choice of knot vectors, the control points exert local control.

A common way to model arbitrarily complex smooth surfaces is to use a mesh of bicubic NURBS patches. Regular meshes consisting of bicubic patches formed by vertices of valence four can be seen as connected planar graphs. A direct consequence of the Euler characteristic for connected planar graphs with the aforementioned properties is that such meshes must be topologically equivalent to an infinite plane, a torus, or an infinite cylinder - all other shapes cannot be constructed unless using trimming or stitching. The resulting surfaces offer precise feature control at the cost of computational complexity due to trimming and stitching [Far99].

Subdivision Surfaces

Subdivision surfaces are the generalization of spline surfaces to arbitrary topology. Instead of evaluating the surface itself, the refinement of the control polygon represents the subdivision surface. There are many different subdivision schemes, e.g., Catmull-Clark [CC78], Doo-Sabin [DS78], Loop [Loo87], Kobbelt [Kob96], etc.

The subdivision scheme presented by Edwin Catmull and Jim Clark is a generalization of bicubic B-Spline surfaces to arbitrary topology [CC78]. The set of 4 x 4 control points p_{ij} forms the starting mesh for an iterative refinement

process where each step results in a finer mesh. The following steps explain the procedure in the local neighborhood of the control points in the regular case (see also [Far02] and [Ull11]):

1. **Face point** For each face in the control mesh, the centroid of its vertices forms a new face point

$$f_{ij} = 1/4(p_{ij} + p_{i+1j} + p_{ij+1} + p_{i+1j+1})$$

2. **Edge point** For each edge in the control mesh, the average of its endpoints and the adjacent face points on either side of the edge form a new edge point

$$e_{ij} = 1/4(f_{ij-1} + f_{ij} + p_{ij} + p_{i+1j})$$

   or

$$e_{ij} = 1/4(f_{i-1j} + f_{ij} + p_{ij} + p_{ij+1})$$

3. **Vertex point** The average

$$Q_{ij}/4 + R_{ij}/2 + S_{ij}/4$$

   forms a new vertex point v_{ij}, where Q_{ij} is the average of the face points of the faces adjacent to the vertex point, R_{ij} is the average of the midpoints of the edges incident on the vertex point, and S_{ij} is the corresponding vertex point.

4. **Create face** A new face consists of a loop of the form

$$f_{ij} - e_{ij} - v_{ij} - e_{ij} - f_{ij}$$

   with f_{ij} closing the loop.

In the regular case, with vertices of valance $n = 4$, the sequence of subdivision steps converges to a limit surface, which is a B-spline patch – irregular cases have no matching B-spline patch.

Subdivision surfaces are invariant under affine transformations. They offer the benefit of being easy to implement and computationally efficient (see the work of Jos Stam on exact evaluation of the limit surface [Sta98]). Only the local neighborhood is used for the computation of new points. Efficiency is not lost even when dealing with arbitrary control meshes along with extraordinary vertices (see Figure 2.4). A major advantage of subdivision surfaces is their repeated refinement process – algorithms can therefore adapt to limited hardware resources often found in mobile devices.

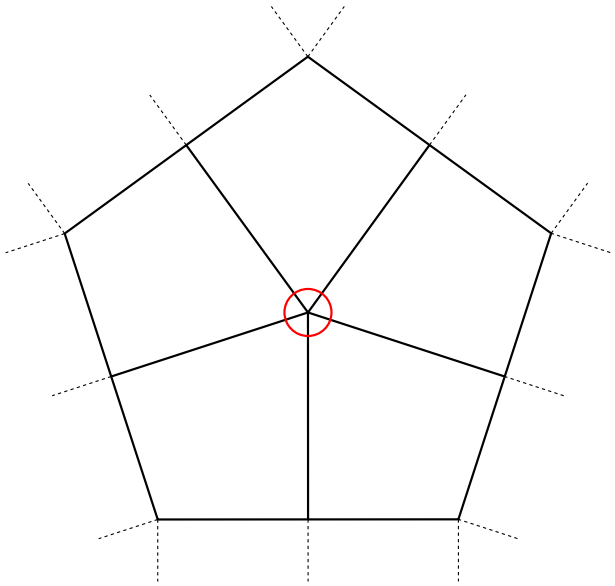


Figure 2.4: A regular mesh of bicubic patches consists of vertices with valance four and is topologically equivalent to an infinite plane, a torus, or an infinite cylinder. To construct different shapes, extraordinary vertices are introduced to the mesh - in this example, a vertex of valence five.

2.4.4 Implicit Surface Representations

In contrast to the parametric surface representations described in Section 2.4.3, implicit surfaces are defined as isosurfaces by a function $\mathbb{R}^3 \rightarrow \mathbb{R}$ [Sul04]. Therefore, similar to voxels, a surface is only indirectly specified. A simple 3D example of an implicit surface is the following definition of a torus with major radius R and minor radius r

$$f(x, y, z) = (x^2 + y^2 + z^2 + R^2 - r^2)^2 - 4R^2(x^2 + y^2) = 0.$$

Inside and outside of the surface are defined by $f(x, y, z) < 0$, respectively $f(x, y, z) > 0$. While a parametric description of the torus exists, many implicit surfaces do not have a closed, parametric form. In terms of expressiveness, implicit surfaces are more powerful than parametric surfaces [KHH+07].

Drawbacks of implicit surfaces are the inherent difficulty of describing sharp features (unless trimming is used) or finding points on the surface. However, this representation has several advantages. Efficient checks whether a point is inside a shape or not are possible. Surface intersections, as well as Boolean set operations (see Section 2.5.3) can also be implemented efficiently. Since the surface is not represented explicitly, topology changes are easily possible.

Implicit surfaces can be described in algebraic form (see the example of the torus), as a sum of spherical basis functions (so called blobby models), as convolution surfaces (skeletons), procedurally, as variational functions, or by using samples. The latter approach directly relates to volumetric shape descriptions and Section 2.5.1.

2.5 Volumetric Shape Descriptions

Volumetric approaches can be used to indirectly describe a shape's surface. In contrast to surface-based descriptions, they define the surface to be a boundary between the interior and the exterior of a shape. However, the idea behind these approaches is not so much a description of a shape's surface, but a description of the entire volume. Such representations are frequently used in visualization and analysis of medical and scientific data.

Algorithmic shape detection on volumetric shape descriptions can be performed similar to what is done for surface-based shape descriptions (see Section 2.4). A surface-based representation can always be obtained from volumetric approaches, and often vice versa.

2.5.1 Voxels

Data sets originating from measurements do not have continuous values and are limited to the points in space where measurements have been collected. It is very common that data points have a uniform regular grid structure. Such data points in 3D are known as voxels, a name related to their 2D counterparts: the pixels. Since a voxel represents only a single point on the grid, the space between voxels is not represented at all. Depending on the area of application, the data points can be multi-dimensional, for example, a vector of density and color. Due to the fact that position and size of a voxel are pre-defined, voxels are good at representing regularly sampled spaces. However, the approximation of free-form shapes suffers from this inherent property. The artwork of Georg Seibert called "Der Käfer - Ein Deutsches Wunder" (see Figure 2.5) is an artistic combination of free-form surfaces and discrete volumetric elements.

Figure 2.5: "Der Käfer - Ein Deutsches Wunder" from Georg Seibert is a memorial for a german miracle - the Volkswagen Beetle. In the context of this thesis it demonstrates the inherent differences of free-form surfaces and discrete volumetric elements.
(Source: Torsten Ullrich, 2011)

Voxel representations do not suffer from numerical instabilities as they are typically defined on an integer grid. A major drawback of voxel representations is the amount of data needed for storage. For example, a $512 \times 512 \times 512$ voxel grid storing 32-bit floating point values occupies 512MB of memory. Depending on the intended use, e.g., in the computer games industry, the memory footprint may be too high and it may appear rather coarse.

Typical use cases are the visualization and analysis of medical data (medical imaging) acquired from sources like Computed Tomography (CT), Magnetic Resonance Imaging (MRI), or 3D ultrasonography. The purpose of these non-invasive imaging techniques is to reveal internal structures of beings that are hidden by skin and bones. High spatial resolution and less noisy data are forming the basis of advances in diagnosis and treatment of diseases.

2.5.2 Convex Polytopes

Shapes can be approximated as geometric objects with flat sides – so called polytopes. They are defined in any dimension as n-dimensional polytopes or n-polytopes. Two-dimensional polygons are called 2-polytopes and three-dimensional polytopes are called 3-polytopes. A special case of a polytope is a convex polytope having the additional property of being a convex set of points in n-dimensional Euclidean space $\mathbb{E}^d$. Convex polytopes can be defined over their convex hull, or by the intersection of half-spaces.

Branko Grünbaum and Geoffrey C. Shephard define a convex polytope as the convex hull of any finite set of points in Euclidean space $\mathbb{E}^n$ ($n \geq 0$) [GS69]. A definition relying on the convex hull of a set of points is called a vertex representation.

Convex polytopes can also be defined as the intersection of a finite number of half-spaces [KUF14]. Because of the fact that the intersection of arbitrary half-spaces need not be bounded, this property must be explicitly required. An algebraic formulation for convex polytopes consists of the set of bounded solutions to a system of linear inequalities. Hence, a closed convex polytope can be written as a system of linear inequalities

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &\leq b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &\leq b_2 \\ \vdots \qquad \vdots \qquad & \quad \vdots \quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &\leq b_m \end{aligned}$$

with m defining the number of half-spaces of the polytope. Open convex polytopes are defined similarly with strict inequalities instead of non-strict ones [TKZ+13b].

A limitation of convex polytopes is the inherent restriction to represent convex geometry only. The representation of non-convex geometry is possible through composition of convex polytopes. Topologically, convex polytopes are homeomorphic to a closed ball.

Convex polytopes are a subject of mathematical study since ancient Greek times. There is a number of special polytopes in three-dimensional space admitting a particularly high degree of symmetry – the so called Platonic solids,

tetrahedron, hexahedron, octahedron, dodecahedron and icosahedron (see Table 2.1). They are bounded by congruent regular polygonal faces exhibiting a consistent vertex valance over all vertices. The five Platonic solids are named after their number of faces.

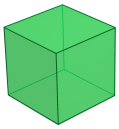
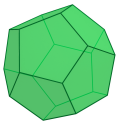
Platonic solids		Vertices	Edges	Faces
tetrahedron		4	6	4
hexahedron		8	12	6
octahedron		6	12	8
dodecahedron		20	30	12
icosahedron		12	30	20

Table 2.1: In three-dimensional space, five convex polytopes exhibit a particularly high degree of symmetry – the so called Platonic solids. Tetrahedron, hexahedron, octahedron, dodecahedron, icosahedron (from top to bottom) are constructed by congruent regular polygonal faces with the same number of faces meeting at each vertex.

2.5.3 Constructive Solid Geometry

Constructive solid geometry (CSG) is a technique to create complex shapes out of primitive objects. These CSG primitives typically consist of cuboids, cylinders, prisms, pyramids, spheres and cones. Complex geometry is created by instantiation, transformation, and combination of the primitives. They are combined by using regularized Boolean set operations like union (denoted by $\cup$), difference (denoted by $\setminus$) and intersection (denoted by $\cap$) that are included in the representation. The union of two sets A and B is the set of elements which are in A, in B, or in both A and B:

$$A \cup B = \{x | (x \in A) \vee (x \in B)\}$$

The intersection of A and B is the set of elements which are in both A and B:

$$A \cap B = \{x | (x \in A) \wedge (x \in B)\}$$

The difference of A and B is the set of elements in B but not in A:

$$A \setminus B = \{(x \in B) | (x \notin B)\}$$

A CSG object is represented as a tree with operators as nodes and primitives as leaves. In contrast to the example in Figure 2.6, not all nodes have to be Boolean set operations, also transformation operations are possible. The example shows the result of the consecutive union of three cylinders subtracted from the difference of a cylinder and a sphere.

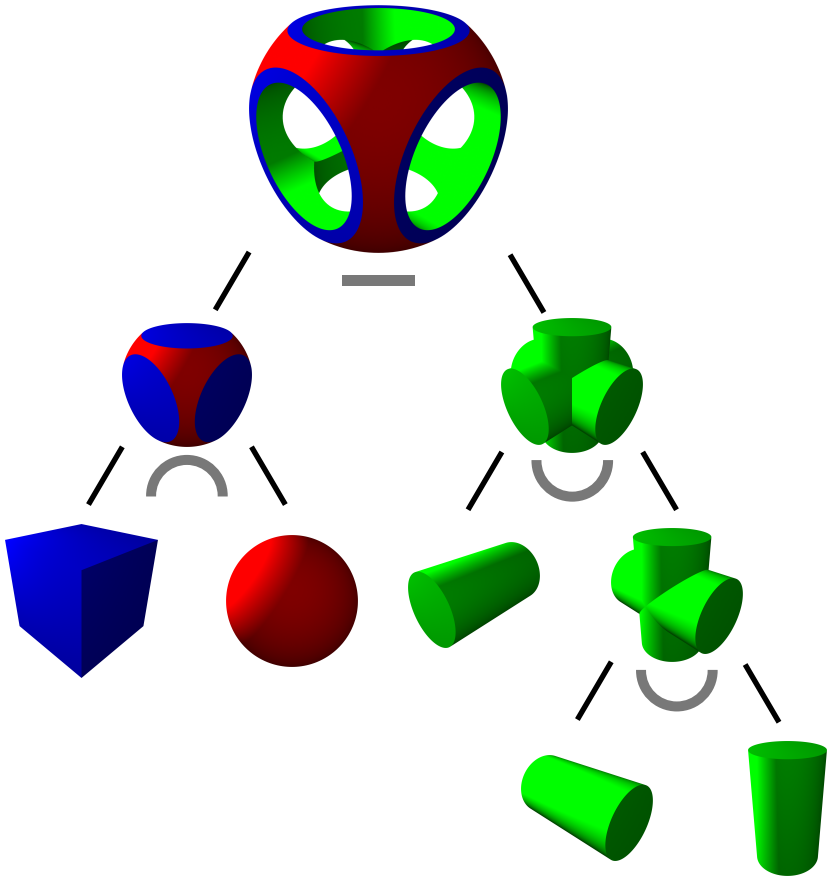


Figure 2.6: The construction of CSG objects can be represented as a binary tree, with leaves as primitives, and nodes as Boolean set operations. Operations are denoted under the nodes. In this example, the consecutive union of three cylinders is subtracted from the difference of a cylinder and a sphere to create the final shape.

In order to determine the shape described by a CSG tree, all operations have to be evaluated bottom-up until the root node is evaluated. Depending on the

representation of the leaf geometry, this task can vary in complexity. Some implementations rely on representations that require the creation of a combined shape for the evaluation of the CSG tree, others do not create a combined representation. In that sense, CSG is not as much a representation as it is a set of operations that need to be implemented for the underlying shape representation [HKS+10].

However, CSG can also be performed on other shapes and shape representations. Two different approaches can be used to create CSG objects: Object-space approaches and image-space approaches. The main difference between the two approaches is that object-space approaches create shapes, while image-space approaches "only" create correct images.

Image-space CSG

Image-space CSG is concerned with creating correct images, used for visualization purposes only. Due to its nature, image-space methods are all view dependent resulting in a re-evaluation of the CSG operations per frame. Since these approaches are generally applicable to be executed on graphics hardware, real-time applications are possible (see Section 3.4.3). Common approaches for image-space CSG are based on scan-line, ray-casting or depth-buffer algorithms.

Scan-line algorithms Scan-line algorithms for CSG are related to classic scan-line algorithms in computer graphics. James D. Foley et al. presented a scan-line algorithm for displaying polyhedral models [FvDFH90]. In principle, the algorithm consists of two parts: a pre-processing part for sorting and a display part. Scan-line algorithms to determine the visibility of a face follow that principle by classifying faces against a CSG tree.

Ray-Casting algorithms CSG can be evaluated in image-space using ray-casting algorithms. For a ray defined by eye position and image plane, the closest visible surface has to be found using intersection tests. This amounts to evaluating the CSG tree for each ray, i.e., on one-dimensional intervals. For each ray, all calculations can be processed in parallel on graphics hardware. Depending on the representation of the objects, the intersection tests can be more or less complex – intersection tests on implicit surfaces (see Section 2.4.4), for example, can be performed efficiently.

Depth-buffer algorithms Jack Goldfeather et al. presented an algorithm, that uses the depth-buffer for evaluating CSG operations [GMTF89]. The algorithm converts a CSG tree to a normalized form. Bounding-box pruning is then performed to allow for an efficient rendering for most CSG objects. In order for this algorithm to work, the parity of a pixel needs to be calculated – it indicates, whether a pixel is inside, or outside a given volume. This can be determined, when all boundaries are closed and all surfaces do not intersect themselves. The depth-buffer is used to detect the pixels that satisfy the CSG logic of the normalized tree.

Object-space CSG

Object-space CSG approaches using primitives described implicitly can be calculated accurately. Performing CSG on other shape representations (like polygonal meshes) typically introduces accuracy problems, due to the finite precision of floating-point numbers. A common representation used for CSG operations are binary space partitioning (BSP) trees. BSP is a method for subdividing a space into convex cells yielding a tree data structure. This data structure can be used to perform CSG operations using tree-merging as described by Bruce Naylor, John Amanatides and William Thibault [NAT90]. The algorithm is relying on accurate information of inside and outside of a shape (or, in case of planes, above and below). Another representation used for CSG operations are half-spaces (see Section 2.5.2).

2.6 Algorithmic Shape Descriptions

Algorithmic shape descriptions are also called generative, procedural, or parametric descriptions. However, there are differences between the three terms. Parametric descriptions are loop-computable programs (the functions it can compute are the primitive recursive functions), and therefore always terminate [Sch08]. On the other hand, procedural descriptions offer additional features, like infinite loops (the functions it can compute are computable functions), are structured in procedures, and are not guaranteed to terminate. Compared to procedural descriptions, generative descriptions are a more general term, including, for example, functional languages.

In the context of this thesis, algorithmic descriptions are generally referred to as generative descriptions. The process of creating such descriptions is referred to as generative modeling. In contrast to many other descriptions, which are only describing a shape's appearance, generative shape descriptions represent inherent rules related to the structure of a shape. In simple terms, it is a computer program for the construction of the shape. It typically produces a surface-based or volumetric shape description for further use, e.g., for visualization purposes. In the article "Modeling Procedural Knowledge – A Generative Modeler for Cultural Heritage" [SSUF10a] by Christoph Schinko et al., we state that all objects with well-organized structures and repetitive forms can be described in such a way. Many researchers enforce the creation of generative descriptions due to its many advantages [KSU15].

Its strength lies in the compact description compared to conventional approaches, which does not depend on the counter of primitives but on the model's complexity itself [BFH05]. Particularly large scale models and scenes – such as plants, buildings, cities, and landscapes – can be described efficiently. Generative descriptions make complex models manageable as they allow identifying a shape's high-level parameters.

Another advantage is the included expert knowledge within an object description, e.g., classification schemes used in architecture, archaeology, civil engineering, etc. can be mapped to procedures. For a specific object only its type and its instantiation parameters have to be identified. This identification is required by digital library services: markup, indexing, and retrieval [FH05]. The importance of semantic meta data becomes obvious in the context of electronic

product data management, product lifecycle management, data exchange and storage or, more general, of digital libraries.

Generative descriptions have been developed in order to generate highly complex shapes based on a set of formal construction rules. They represent a whole family of shapes, not just a single shape. A specific exemplar is obtained by defining a set of parameters, or a sequence of processing steps: Shape design becomes rule design [KSU14].

Because such descriptions already belong to a specific class of shapes, there is no need for detectors. However, with a generative description at hand, it is interesting to enrich other descriptions and representations. What is the best generative description of one or several given instances of an object class? This question is regarded as the inverse modeling problem and is discussed in Chapter 5.

2.6.1 Characterization

Generative descriptions are typically characterized using techniques for the description of formal languages and compiler construction [Par10]. The range of different language concepts used to create generative shape descriptions is very wide and consists of different linguistic concepts [Cho56]. In the article "Built by Algorithms – State of the Art Report on Procedural Modeling" [SKUF15] by Christoph Schinko, Ullrich Krispel, Torsten Ullrich and Dieter W. Fellner, we follow a structured approach. The main categories to describe a shape are

- **rule-based**: using substitutions and substitution rules to build complex structures out of simple starting structures [ÖK08], [KPK10], [SK92].
- **imperative and scripting-based**: using a scripting engine and techniques used in predominant programming languages [Hav05], [SSUF11b], [KK11], or
- **GUI and dataflow-based**: using new graphical user interfaces (GUI) and intelligent GUIs to detect structures in modeling tasks, which can be mapped onto formal descriptions [TKHF12].

The general principles of formal descriptions and compiler construction are the same in all cases, independent of ahead-of-time compilation, just-in-time compilation or interpretation (see Chapter 4).

Rule-based Systems

Rule-based systems are using a representation that has proven to be useful for generative modeling. Such systems provide a declarative description of the construction behavior of a model by a set of rules. In the context of shape descriptions, shape grammars and especially split grammars are applicable. Shape grammars use a generation engine to select and process shape rules. Shape rules define how an existing shape can be transformed. A particular kind of shape grammar, called a split grammar consists of split operation, which decompose a shape into a set of smaller shapes. Systems using these kind of grammars are primarily used to construct buildings or façades.

Imperative Systems

In many cases, classical programming paradigms are used for generative descriptions: Commands to generate a specific shape are written in a programming language using a library that utilizes some sort of geometry description and operations to perform changes. The commands are issued in the form of statements forming the program. Typically a program is built from one or more procedures. Note that the resulting geometry is often produced as a side effect. Furthermore, any modeling software that is scriptable by an imperative language or provides some sort of Application Programming Interface (API) falls into this category.

Dataflow-based Systems

In dataflow-based systems, the program is represented by a directed graph of the data flowing between operations. The movement of data is emphasized and programs are modeled as a series of connections - implying the use of a graphical representation. These systems are called Visual Programming Languages (VPL) and allow to create a program by linking and modifying visual elements. Although many VPL's are based on the dataflow paradigm, also other paradigms like flow charts are suitable.

2.7 Summary

This chapter focused on defining the term shape and providing a structured introduction to different ways of describing and representing shapes. Depending on the actual description, the perception (or detection) of shapes shows differences between human beings and computers. While textual and image-based descriptions are of more abstract nature (for a computer), geometry-based descriptions of shapes offer a more formal, low-level approach. Algorithmic shape descriptions try to combine the advantages of a high-level approach (being more suitable for humans) with the benefit of not being too abstract for computers. The choice of description, however, also has direct effects on possible shape modeling paradigms. These will be discussed in the next chapter with a special focus on generative modeling.

Chapter 3

Modeling

Based on the shape descriptions presented in the previous chapter, this chapter focuses on modeling paradigms. The computer graphics term shape modeling stands for the process of developing a mathematical representation of an object using specialized software. Shape modeling can be done in many ways, however the traditional ways of shape modeling (e.g., clay modeling, sculpting) are not in the focus of this this chapter.

The next three sections give insight into primitive modeling using 3D modeling software or scene description languages, semantic modeling dealing with meta data, and generative shape modeling using domain specific information. Then, an application for generative modeling – wedding rings – is presented using the Generative Modeling Language (GML). Finally, an approach for delivering this generative content over the web in a controlled environment with a focus on intellectual property protection concludes this chapter.

Contents

3.1 Primitive Shape Modeling

Many shape descriptions mentioned in Chapter 2 are used in software products for primitive shape modeling (e.g., surface-based and volumetric shape descriptions). Modeling software greatly differs in the level of abstraction – depending on the domain – as well as in the user interface.

3.1.1 3D Modeling Software

Dedicated programs like Autodesk Maya, Autodesk 3DS Max, or Blender work with virtual workspaces (scenes). Scene elements are node-based with each node having its own attributes. Components for different tasks like polygonal modeling, fluid effects or particle systems are accessible through a visual workflow. The user interface is typically adapted to the specific task, however the overall screen layout seldomly changes. Figure 3.1 shows the default screen layout of Blender offering five editors.

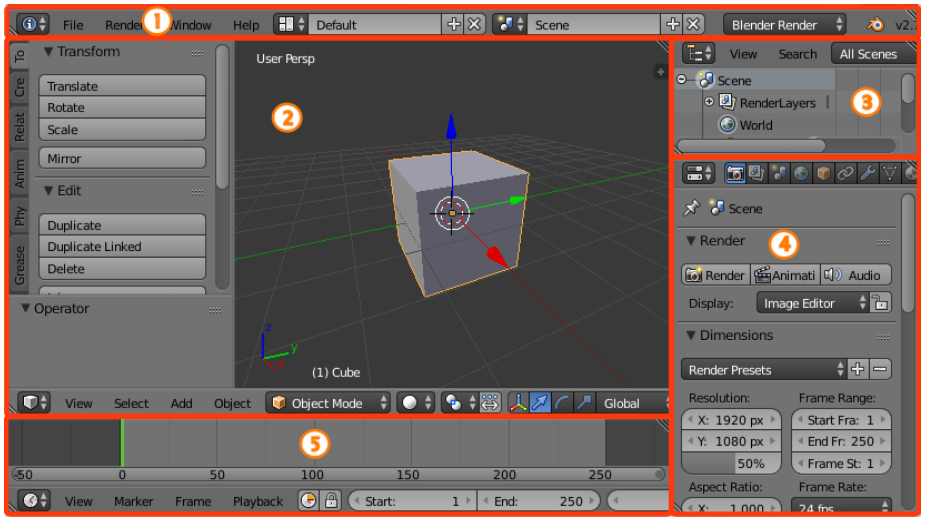


Figure 3.1: The default window of Blender appears. It offers five editors: Info (1), 3D View (2), Outliner (3), Properties (4) and Timeline (5).
(Source: Blender Foundation, 2015)

These dedicated programs are typically equipped with scripting possibilities. Apart from the accessibility of features through the user interface, some functionality may not yet be equipped with an appropriate interface, or is more conveniently used programmatically (in fact, the user interface is often directly built upon the scripting language). A scripting language may also provide additional functionality not available through the user interface. The spectrum of scripting languages ranges from simple and highly domain-specific languages to general-purpose programming languages.

Powerful scripting languages offer means to customize and add functionality, as well as to speed up complicated or repetitive tasks. Common languages used in modeling software are Python (Blender, Maya), MaxScript (3DS Max), and Maya Embedded Language (Maya).

Data Acquisition and Shape Reconstruction

The reconstruction of shapes from sensor data (often also called scanning) is necessary, wherever real-world designs (e.g., clay models) are used as part of an otherwise digital workflow. Especially in the context of automotive design, real-world models are still an integral part, due to the possibility for designers to use their tactile senses, rather than sight alone.

Range imaging techniques produce images with pixel values that correspond to the distance from a specific point. Data acquisition is concerned with generating a point set out of a number of range images. The possibilities to acquire range images are manifold. These techniques are typically associated with some sort of sensor device (cameras). Some common approaches for capturing range images are:

- **Stereo triangulation**: The depth values of pixels are determined from data acquired using a stereo or multiple-camera setup. This method relies on corresponding points in the multitude of images. Since it is a difficult problem to solve in homogeneous regions, quality and quantity of the acquired points heavily depends on "good" images.

- **Structured light**: Different light patterns can be used to determined depth using only a single image of the reflected light. Reflective or transparent surfaces cause problems during detection. However, research by Mohit Gupta et al. to handle optically complex scenes by redesigning illumination patterns shows promising results [GAVN11].

- **Time-of-flight**: By measuring the time a light pulse takes to get reflected by an object, it is possible to capture range images. Background light, reflections from specular surfaces, and interference can cause problems for time-of-flight cameras.

- **Triangulation**: Triangulation-based approaches use laser light to capture the environment. Depending on the distance to the object, the laser point appears at different points in the field of view of the the measuring device. This approach is called triangulation because the laser spot, the camera and the laser emitter form a triangle.

The creation of a shape from point set data is called shape reconstruction. However, scanned data often contains a wide variety of defects, like missing parts, or points from incorrectly matched features. The survey of Matthew Berger et al. provides an overview of the current state of the art [BTS+14].

3.1.2 Scene Description Languages

In contrast to the visual workflow, or the scripting possibilities of dedicated programs used for scene modeling, scene description languages offer a text-based approach to describe a scene. The nature of scripting languages is imperative, or procedural, whereas the nature of scene description languages is declarative.

Some scene description languages include variables, constants, conditional statements, and loops. These descriptions are used in modeling software to store scenes. While there are many vendor-specific binary formats available for storing scenes space-efficiently, a scene stored in a description language offers the advantage to be both human-readable and machine-readable – for this purpose the Extensible Markup Language (XML) is commonly used.

Extensible 3D (X3D) is a royalty-free XML-based declarative approach to describe scenes [Web08]. It is successor to the Virtual Reality Modeling Language (VRML) and includes a large number of new and extended features and components (e.g., Humanoid Animation, NURBS, GeoVRML, ...) [VRM97]. The X3D specification includes numerous APIs and a full runtime, event and behavior model. Several sets of components for various levels of capability include X3D Core, X3D Interchange, X3D Interactive, X3D CADInterchange, X3D Immersive, and X3D Full.

As X3D is designed to be integrated into HyperText Markup Language (HTML) pages, two projects propose plugin-free implementations. The XML3D project proposes an extension to HTML5 for describing interactive 3D scenes allowing seamless integration of 3D content into HTML5 pages [SKR+10]. On the other hand, the X3DOM project allows including X3D elements as part of any HTML5 Document Object Model (DOM) tree [BEJZ09]. The goal here is to have a live X3D scene in the HTML5 DOM, which allows manipulation of the 3D content by adding, removing, or changing DOM elements. In an attempt to integrate interactive 3D graphics capabilities into the World Wide Web Consortium (W3C) technology stack, members of both projects formed the W3C community group "Declarative 3D for the Web Architecture"[1].

Scene description languages are also directly used in tools like POV-Ray – without a scripting layer, or visual workflow – to generate images. For example, Figure 3.2 (left) shows a POV-Ray script defining a simple scene: background color, camera, light source and geometry. The resulting output is a green torus, as can be seen on the right side of the figure.

3.2 Semantic Modeling

Generally, meta data is a summary of basic information about data, simplifying the process of finding and working with particular instances of data. Basic examples for document meta data are *author*, *date created* and *date modified*. With the ability to filter through meta data, it is much easier to locate documents. This semantic information is not limited to textual documents, but documents in general (images, videos, web pages, shapes, ...). Semantic modeling – in this context – is concerned with enriching shapes with meta data.

Primitive modeling as well as automatic shape reconstruction suffer from a lack of semantic information. A collection of any primitives inherently offers a small amount of meta data. The enrichment with (and the explicit representation of) semantic information is vital in the context of digital library services: indexing, archival, and retrieval. In the article "Semantic Enrichment for 3D Documents: Techniques and Open Problems" [USB10] by Torsten Ullrich, Volker Settgast and René Berndt, they classify meta data according to the following criteria:

[1] https://www.w3.org/community/declarative3d/

```
#version 3.7;
#include "colors.inc"

// background color
background {color rgb <1, 1, 1>}

// camera
camera {location  <0.0, 0.0, -5.0>
        direction 1.5*z
        right     x*image_width/image_height
        look_at   <0.0, 0.0, 0.0>}

// light source
light_source {<0, 0, 0>
              color rgb <1, 1, 1>
              translate <-10, 10, -10>}

// torus
torus {1, 0.5
       rotate  -45*x
       texture {pigment {Green}
                finish  {specular0.6}}}
```

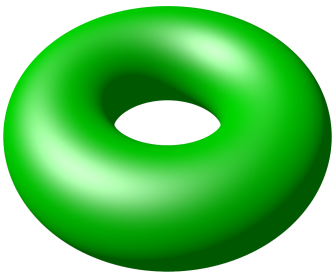


Figure 3.2: The text-based description on the left is a POV-Ray script defining a simple scene: background color, camera, light source and geometry. A rendering of the resulting scene – a green torus – is on the right.

Data Type The data type of the object can be of any elementary data structure (e.g., Polygons, NURBS, Subdivision Surfaces, ...).

Scale of Semantic Information This property describes, whether meta data is added for the entire data set or only for a sub part of the object.

Type of Semantic Information The type of meta data can be descriptive (describing the content), administrative (providing information regarding creation, storing, provenance, etc.) or structural (describing the hierarchical structure).

Type of creation The creation of the semantic information for an object can be done manually (by a domain expert) or automatically (e.g., using a generative description).

Data organization The two basic concepts of storing meta data are storing the information within the original object (e.g., EXIF data for images), or storing it separately (e.g., using a database).

Information comprehensiveness The comprehensiveness of the semantic information can be declared varying from low to high in any gradation.

Having classified different types of meta data, many concepts for encoding semantic information can be applied to 3D data. Despite the large number of 3D data formats, only a few are standardized, non-proprietary and support semantic markup [Set13]:

COLLAborative Design Activity (Collada) The XML-based Collada format is an ISO standard and allows storing meta data like title, author, revision etc. not only on a global scale but also for parts of the scene [Int12a]. This file format can be found in Google Warehouse where meta data is, for example, used for geo-referencing objects.

Initial Graphics Exchange Specification (IGES) IGES is an ANSI standard since 1980 and allows the definition of annotations including dimensioning data as well as labels and notes [U.S06]. This file format is used as a vendor-neutral exchange format among CAD systems.

Jupiter Tesselation (JT) The JT file format is an ISO standard since 2012 and is used for product visualization and data exchange in CAD systems [Int12b]. Annotations in the form of attributes and properties as well as filters are supported by this format. It is accompanied by the XML-based format PLMXML to represent product structure hierarchy.

Portable Document Format (PDF) 3D PDF 3D is an ISO standard and allows to store annotations separated from the 3D data even allowing annotating the annotations [Int08]. An advantage is that the viewer application is widely spread and PDF documents are the quasi standard for textual documents.

Standard for the Exchange of Product model data (STEP) STEP, an ISO standard since 1994, is divided into different parts, data models and environments [Int94]. The current Application Protocol 242 supports product data and non-geometrical meta data.

Extensible 3D Graphics (X3D) The X3D file format is an XML-based ISO standard for representing 3D computer graphics [Int13]. It supports a number of different meta data nodes providing arrays of strongly typed data.

While a standard has advantages for accessibility, long-term archival, and many other aspects, it does not solve inherent problems; i.e., due to the persistent naming problem, a modification of the 3D model can break the integrity of the semantic information. Any change of the geometry can cause the referenced part of the model to no longer exist or being changed. Nevertheless, there are a lot of examples for semantic modeling in various contexts.

Alexandre Boulch et al. propose an approach to automatically enrich complex objects in a 3D scene with semantic information [BHMT13]. A constrained attribute grammar is used to automatically compute a shared parse forest of all interpretations. As a parse tree reflects the structure of a 3D scene, it enriches scene primitives with semantic labels and relations.

In their generative modeling approach, Simon Haegler, Pascal Müller and Luc Van Gool produce multiple models to sample the space of possibilities introduced by uncertainty in the context of digital cultural heritage [HMVG09]. Missing knowledge about the past appearance of archaeological sites is typically only visualized using transparency, or different textures. The inherent structural description encoded in generative models explicitly expresses the uncertainty.

Erick Mendez et al. present a generative modeling pipeline to create interactive 3D visualizations of underground infrastructure [MSH+08]. The 3D models are encoded in a scene-graph representing visual models with semantic markup. It is used for interactive filtering and styling of the models in an augmented reality setup.

In their work, Liu Yong et al. identify three challenges of generative modeling in an architectural context: complex architecture, efficient manual creation, automatic semantic enrichment [YMYH12]. They address these challenges with a general framework to construct large-scale 3D models of digital architectural heritage.

3.3 Generative Modeling

Generative modeling techniques are used to generate highly complex shapes based on a set of formal construction rules. In Section 2.6 we have established, that this modeling paradigm describes a shape by a sequence of processing steps, rather than just the end result of applied operations. Since generative modeling is a very general approach, it can be applied to any domain and to any shape description that provides a set of generating functions. Another advantage is the included expert knowledge, e.g., classification schemes used in architecture, archaeology, civil engineering, etc. can be mapped to generating functions. For a specific object only its type and instantiation parameters have to be identified. As mentioned in Section 3.2, this semantic information is required by digital library services. In the article "Know the Rules – Tutorial on Procedural Modeling" [SKU15] by Christoph Schinko, Ulrich Krispel, and Torsten Ulrich, we describe the following fields of application for generative modeling techniques.

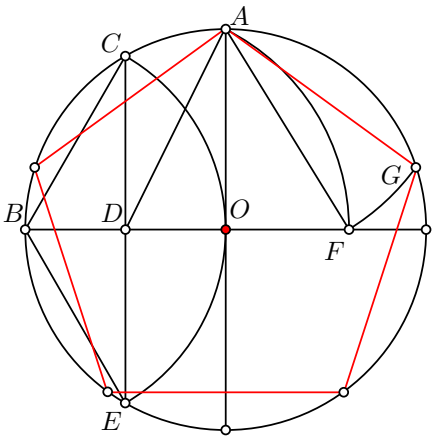
3.3.1 Ruler and Compass

The construction of lengths, angles, and other geometric figures using only an idealized ruler and compass are called ruler-and-compass constructions. The idealized ruler is assumed to be of infinite length, offering only one edge with no markings on it. The compass, on the other hand, is not able to be directly used to transfer distances. Ancient Greek mathematicians placed great emphasis on problems of constructing various geometric figures using only ruler and compass. Ruler-and-compass construction problems that they could not overcome, and that were later shown to be impossible to perform, were thought to be hard, not unsolvable. However, the ancient Greek knew how to solve many of these problems without the constraints of working only with ruler and compass.

All ruler-and-compass constructions consist of repeated application of five basic constructions based on Euclid's axioms [Euc07] relying on the points, lines and circles that have already been constructed. An example of a ruler-and-compass construction is illustrated in Figure 3.3. These geometric primitives in combination with a fixed set of operations are the first algorithmic descriptions of generative models.

Since Greek geometric constructions are prominently described in Euclid's Elements [Euc07], these constructions are sometimes also referred to as Euclidean constructions. Many geometric problems of antiquity like circle squaring,

cube duplication, and angle trisection are closely related to these constructions. Although the Greeks were unable to solve these problems, constructions for regular triangles, squares, pentagons, and their derivatives had been given by Euclid.



Construction of a pentagon:

1. Draw a circle in which to inscribe the pentagon and mark the center point O.
2. Construct a pair of perpendicular lines, which intersect in O and mark their intersection with the circle A and B.
3. Let D be the midpoint of $\overline{BO}$. The circle with center D and radius $|\overline{DA}|$ intersects the line defined by the points B and O. Mark the intersection point as F.
4. The length of section $\overline{AF}$ is equal to the edge length $\overline{AG}$ of an inscribed pentagon (red).

Figure 3.3: The construction of a pentagon can be performed using ruler and compass alone. The construction algorithm is based on Euclid's axioms.

As an interesting side note, Georg Mohr stated that all constructions possible with a compass and ruler can be done with a compass alone, as long as a line is considered constructed when its two endpoints are located. The reverse is also true. The Swiss mathematician Jakob Steiner proved that all constructions possible with ruler and compass can be done using a ruler alone, as long as a fixed circle and its center have been drawn beforehand. Such a construction is known as a Steiner construction.

The long history of geometric constructions [Mar98] is also reflected in the history of civil engineering and architecture [Mit90]. In Gothic architecture, for example, quite complex geometric shape configurations can be exhibited. Even if the perception of these shapes leads to believe in a high degree of complexity, it is achieved by combining a relatively small number of basic geometric patterns. Sven Havemann and Dieter W. Fellner present some principles of this long-standing domain. They show how the constructions of a few prototypical Gothic windows can be formalized using generative modeling techniques [HF04]. Complex configurations can be obtained through a combination of elementary constructions by using a modularization of inherent properties of Gothic windows. Different combinations of specific parametric features can be grouped together, leading to the concept of styles. They enable to differentiate between the basic shape and its appearance, i.e., in a particular ornamental decoration (see Figure 3.4) [TKZ+13a]. This concept leads to an extremely compact representation for a whole class of shapes [BFH05].

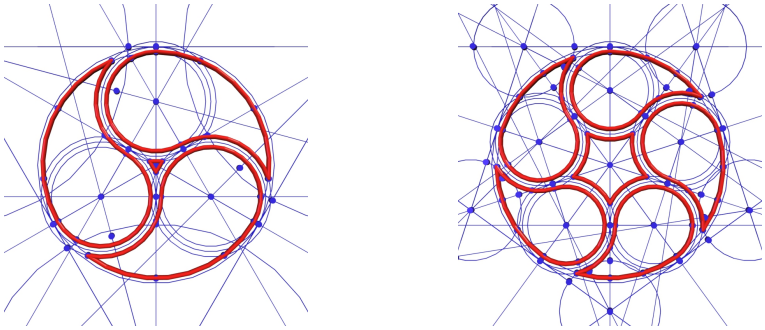


Figure 3.4: Compass and ruler operations have long been used in interactive generative modeling. This Gothic window construction was created in the framework presented by Wolfgang Thaller et al. relying on direct manipulation of the construction alone, without the use of code or graph editing [TKZ+13a].

3.3.2 Architecture

The possibilities of including expert knowledge within a generative object description can, for instance, be exploited by using a mapping from classification schemes to procedures. Depending on the intended use of the description, the degree of abstraction can vary. Building blocks can be created at different levels of abstraction starting at a very low level defining intricate details and culminating at a very coarse level exhibiting parameters with large-scale effect. For a specific object, only its type and its instantiation parameters have to be identified. The generative building blocks themselves are static and do no change. As a consequence, only their parameters have to be specified: Figure 3.5 illustrates combinations of the same building blocks of Gothic architecture.

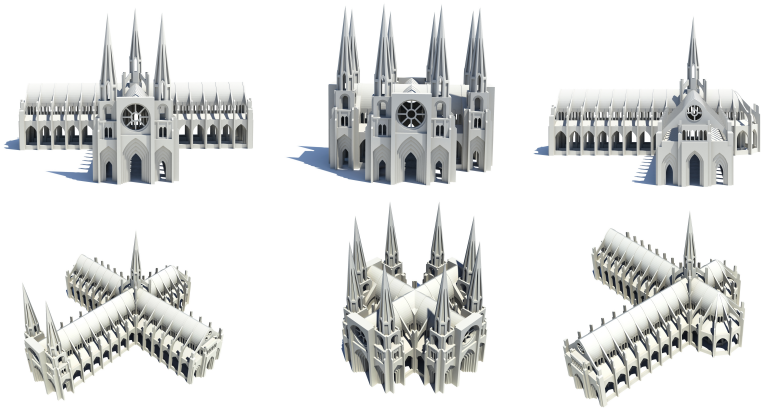


Figure 3.5: These building blocks of Gothic architecture have been combined in various ways to create churches and cathedrals. The generative description takes a few high-level parameters to generate a family of Gothic buildings. Building blocks by Michael Curry, http://www.thingiverse.com/thing:2030.

The building blocks of Gothic architecture have been combined in various ways to create churches and cathedrals. The generative description of Gothic cathedrals encodes these building blocks and the rules on how to combine them. The result is an algorithm that takes a few high-level parameters. The usage of generative modeling techniques in architecture is not limited to buildings of the past where the purpose of such techniques is to reconstruct, not to construct. Over the last few decades, many architects have used a new class of design tools with support for generative design. Generative modeling software enables new design paradigms and extends the design abilities of architects by harnessing computing power in new ways. Computers have long been used to capture and implement the design ideas of architects by means of CAD and, more recently, 3D modeling. But generative design actually helps architects by using computers to extend human abilities.

A very impressive example for generative design is the National Stadium in Beijing (see Figure 3.6). Its main structure is an elliptic steel structure extending 333m from north to south and 294m from east to west, with a height of 69.2m. The stadium has two independent structures, a concrete seating bowl and the outer steel frame around it at about 15m distance. The unconventional structure was designed by Herzog & De Meuron Architekten, ArupSport and the China Architecture Design and Research Group applying generative techniques to speed up the design process.

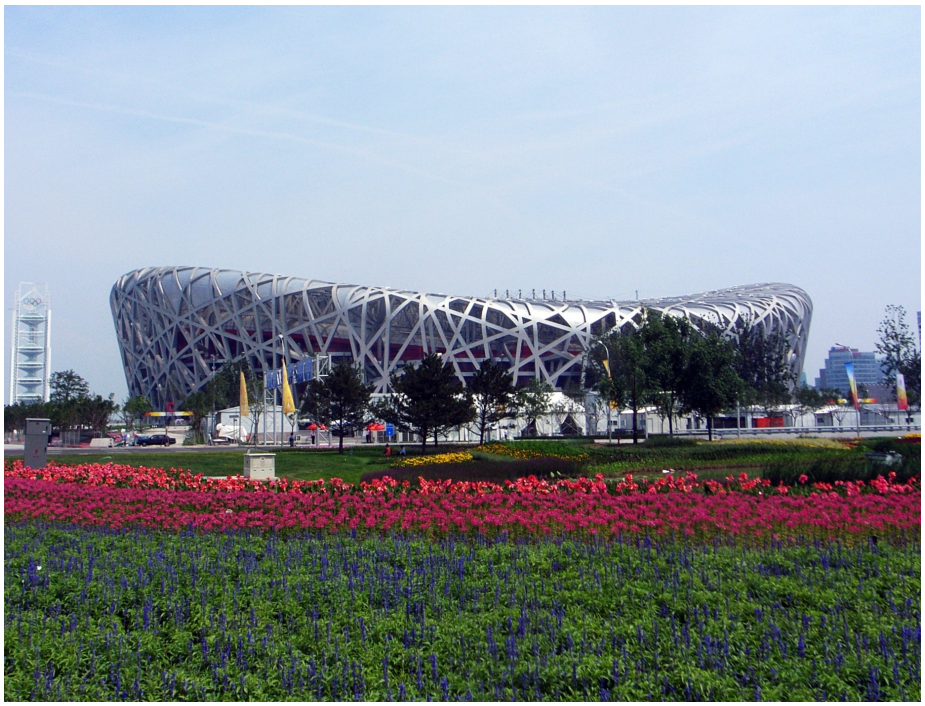


Figure 3.6: The Beijing National Stadium staged the 2008 Olympic Games from 8 August to 24 August 2008. The generative design of its interwoven façade helped saving time in resolving iterations compared to conventional CAD techniques.
(Source: Chen Zhao, 2008 https://flic.kr/p/5iQi1p).

Lars Hesselgren explains generative design with the following words: "Generative design is not about designing a building. It's about designing the system that designs a building."

CGA Shape, CityEngine

Despite the fact that the concept of L-systems (see Section 3.3.4) has been introduced a few decades ago, it is still used nowadays in a more or less unaltered form. In combination with shape grammars, L-systems find their application in procedural modeling of cities [PM01]. Yogi Parish and Pascal Müller presented a system that generates a street map including geometry for buildings given a number of image maps as input. The input consists of, but is not limited to elevation maps, land/water/vegetation boundary maps and population density. To allow the definition of global objectives as well as local constraints, L-systems have been extended. Setting and modification of parameters is transferred to external functions so that every time a rewrite rule is applied, the result of the L-System has no assigned parameters. They are assigned and modified according to global goals and local constraints in subsequent steps leading to reduced complexity and easy extensibility of the rules.

Once the street map is generated, the system creates the allotments for placing the buildings and then generates the geometry as well as a procedural texture. However, the use of procedurally generated textures to represent façades of buildings often results in a limited level of detail.

In later work, Pascal Müller et al. describe a system [MZWVG07] to create more detailed façades based on single façade images and a split grammar called CGA shape. The façade images are subdivided into high-level elements like floors and tiles by detecting repetitions. Low-level structures like windows and doors are detected by a subdivision scheme using algorithms based on knowledge of the architecture of the façades. As a last step, shape grammar rule sets are automatically inferred to allow interactive fine-tuning of the façades.

A framework called the *CityEngine* provides the modeling environment for CGA shape. The framework is seen as useful add-on to accelerate modeling of façades. Yet there are some limitations in the automatic processing stage used to detect the structure of the façades. It does have difficulties with varying quality of the input images, as well as with buildings composed of non-repetitive structures.

Markus Lipp, Peter Wonka and Michael Wimmer present another procedural modeling approach for architecture [LWW08] following the notation of Müller [MWH+06] that emphasizes on real-time interactive visual editing of the underlying grammar. A set of visual operators for rule and building editing as well as tools for local modifications and semantic selections are introduced. The system allows to store local changes persistently over global modifications by using semantic annotations. A visual editing approach has proven to be much more intuitive compared to textual editing of split rules by combining procedural modeling techniques and standard 3D modeling paradigms. Some deficiencies of the system arise from CGA shape itself. The approach is limited to modeling buildings and there is no direct support for curved surfaces.

Model Graphs

Björn Ganster and Reinhard Klein propose a procedural modeling approach [GK07] based on structure trees. They describe an integrated framework relying on a visual language. The infix notation of the language requires the use of variables, which are stored on a heap to circumvent implementing pipelines to transport data. As a consequence, directed edges define the order of execution and special nodes perform variable assignments. The graph structure represents the rules used to create an object. Special nodes allow the creation of geometry, the application of operators as well as the usage of control structures. Various attributes can be set for nodes used in a graph. However, since model graphs are an interpreted language, their performance is not comparable to a compiled application. With increasing complexity, it becomes difficult to keep an overview of the graph, even if model graphs can be distributed into smaller modules. This framework can be used to model trees, buildings and landscapes.

Hierarchical Description

Dieter Finkenzeller presented an approach for detailed building façades based on a parameterization of the whole façade called ProcMod [Fin08]. It features a hierarchical description for an entire building. In order to create a building, the user provides a coarse outline by defining floor plans for each level of the building. Important parts of the building like balconies, cornices and oriels must be defined in this stage. After selecting the basic style of the building, the system generates a graph representing the building. The style is not related to the building outline and can be set independently. In the next step, the system combines building outline with style and creates a hierarchical description represented by a typed graph. This graph is traversed and geometry for every element of the graph is generated. The result is a detailed scene graph, in which each element can be modified afterwards. However, there are some limitations in this approach. Only common building structures and styles are supported. Organic structures and inclined walls cannot be modeled.

3.3.3 Civil Engineering

The generative modeling approach can be applied to any domain and is not restricted to shape representations [CSS+11]. In the context of 3D computer-aided design (CAD), design processes involving repetitive tasks are perfectly suited for a generative approach. In the field of engineering processes can be differentiated in repetitive and creative processes. While all processes are often limited by imposed constraints (e.g., interfaces to other parts, structural soundness), repetitive ones consist of nearly identical tasks and are therefore independent of creative decisions. This condition is necessary for modeling them in a system of rules as demonstrated by Gerald Frank [FH12]: Liebherr manufactures and sells an extensive range of products including ship-, offshore and harbor mobile cranes as well as hydraulic duty cycle crawler cranes and lift cranes (see Figure 3.7).

Each crane has to be partially or fully customized to a customer's needs, but the design process of ascent assemblies is based on a set of invariant rules that can be modeled and stored. In numerous interviews with engineering experts

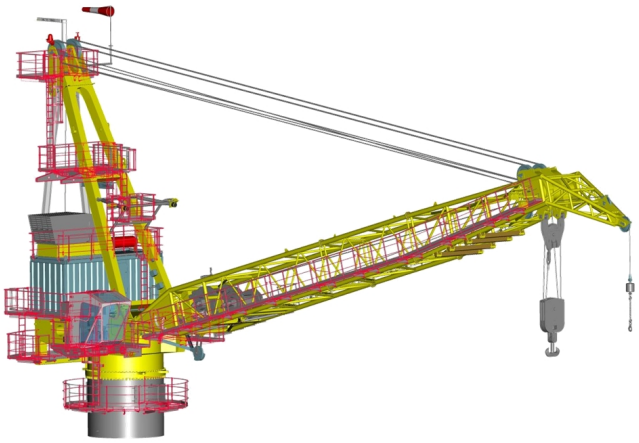


Figure 3.7: The design of ascent assemblies for offshore cranes (colored in red) results in high efforts and causes a major part of the overall engineering costs of a crane. Because of repetitive and nearly identical design processes, the product development processes has been optimized by software driven design automation: the reduction of engineering efforts by modeling design knowledge [Fra11].

at Liebherr the repetitive design processes have been analyzed and a generative model has been designed. Integrated into the existing CAD pipeline, a construction engineer now only has to determine the defining parameters of an assembly and fill out the corresponding input fields in a user interface. The engineering of ascent assemblies of an offshore crane required up to 150 hours. Using the procedural approach, the efforts have been reduced down to ten percent.

3.3.4 Nature

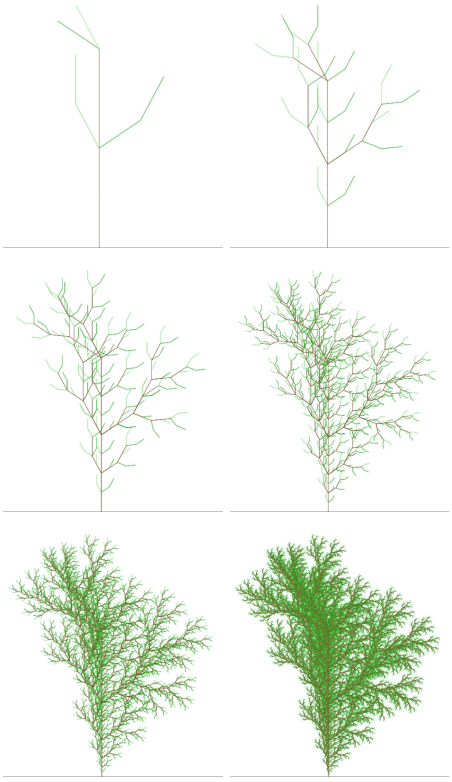
The creation of natural phenomena such as trees, bushes, or flowers is a challenging task. Despite their structural complexity, many organic forms develop following simple rules. On the one hand, the development of plants is determined by their environment (e.g., limited water supply, exposure to the elements, sunlight coming from a specific direction). Plants have different strategies to deal with these limitations. On the other hand there is a plant's inherent hierarchical structure describing its geometric properties. These two aspects (and probably many more, e.g., development over time) have to be combined in a procedural system for the creation of plants.

L-Systems

In today's procedural modeling systems, scripting languages and grammars are often used as a set of rules to achieve a description. From a historical point of view, the first procedural modeling systems date back to the year 1968. At that time, Lindenmayer systems [PL90], or L-systems for short were introduced and developed by the Hungarian theoretical biologist and botanist Aristid Lindenmayer. These systems were conceived to establish a formal

description of the development of simple organisms as well as to illustrate neighborhood relationships between cells of such organisms. While describing rather simple structures in the beginning, these systems were successfully applied to modeling more complex organisms.

An L-system is best-described as a variant of a formal grammar. An alphabet of symbols can be used to create simple strings. More complex strings are created by using a set of string rewriting rules. A predefined set of rules is applied to an initial string forming a new, possibly larger string. The recursive nature of the L-systems approach reflects its biological motivation. However, since a grammar only describes the syntax of a string, not its semantical meaning, an interpretation of the generated strings is necessary in order to generate images or to model geometry. The modeling power of these early geometric interpretations of L-systems was limited to creating fractals and plant-like branching structures (see Figure 3.8).



L-System:

- Axiom: FX
- Angle: 28°
- Rules:

  $F \mapsto C_0FF - [C_1 - F + F] + [C_2 + F - F]$

  $X \mapsto C_0FF + [C_1 + F] + [C_3 - F]$

whereas F denotes "draw forward" and $+/-$ denote "turn left"/"turn right". The square bracket [corresponds to saving the current values for position and angle, which are restored when the corresponding square bracket] is executed. C_0, C_1, C_2 switch colors and X does not correspond to any drawing action.

Figure 3.8: Lindenmayer systems are a simple but elegant "turtle rendering" platform. The recursive nature of L-system rules lead to self-similarity and thereby fractal-like forms. Plant models and natural-looking organic forms "grow" and become more complex by increasing the iteration level i.e., the number of substitutions.
This example can be executed online by Kevin Roast's L-Systems-Demo, http://www.kevs3d.co.uk/dev/lsystems

Later on, parametric L-systems were introduced out of the necessity to overcome issues when dealing with continuous phenomena. For example, discretizing continuous values may require a large number of quantization levels and would lead to a large number of symbols, or even worse, it may not be possible at all. The idea of parametric *L-Systems* is to associate numerical parameters with L-system symbols and thus allowing parameters to control the output - the generative aspect is introduced.

Combined with additional 3D modeling techniques, L-systems can be used to generate complex geometry [TMW02a], [TMW02b]. In order to generate models of plants, terrains, and other natural phenomena that are convincing at different scales, Robert F. Tobler, Stefan Maierhofer and Alexander Wilkie introduce a combination of subdivision surfaces, fractal surfaces, and parameterized L-systems making it possible to choose, which of them should be used at each level of resolution. Since the whole description of such multi-resolution models is procedural, their representation is very compact and can be exploited by level-of-detail renderers that only generate surface details that are visible.

This kind of data amplification can be found in various fields of computer graphics – e.g., curved surfaces specified by a few control points are tessellated directly on the GPU. This results in low storage costs and allows generating the complex model only when needed, while also reducing memory transfer overheads. Although L-systems are parallel rewriting systems, derivation through rewriting leads to very uneven workloads. Furthermore, the interpretation of an L-system is an inherently serial process. Thus, L-systems are not straightforwardly amenable to parallel implementation. In 2010, Markus Lipp, Peter Wonka and Michael Wimmer presented a solution to this algorithmic challenge [LWW10].

Model Graphs

Structure trees, as described in the section about architecture, can also be applied to create natural patterns. Bernd Lintermann and Oliver Deussen proposed a procedural modeling method as well as a graphical user interface for the creation of natural branching structures based on this concept [LD98]. The system represents the modeling process with a structure tree, which can be altered using specialized components describing geometry as well as structure. Components can also be used for defining global and partial constraints. These components are described procedurally using creation rules, which may include recursion. The modeling workflow basically consists of combining components in a structure tree. Geometric data is generated according to the structure tree via a tree traversal, where the components generate their geometrical output themselves. Problems in the modeling process arise through the large amount of parameters and through insufficient understanding of their influence in the component hierarchy.

3.3.5 Entertainment

The *demoscene* is a computer related subculture that specializes in producing *demos*, which are programs that perform audio-visual presentations. Individuals are actively participating in the scene since the advent of personal computers (see the book of Tomas Polgar for a brief history [Pol05]).

The creation of such programs is often tied to various constraints, as memory and general hardware capabilities of the early personal computers were quite limited. Nowadays, these constraints are mostly of self-imposed nature, e.g., creating a demo using only a file size of 4096 bytes (or 4 KB). Naturally, these constraints facilitate the usage of procedural methods for any type of presentation content: 3D geometry (meshes), textures [EMP+02], sound, etc.

Even without these limitations, content creation [TYSB11] has always been the main field of application for procedural techniques: from game design [AA14], [CM06], [JTSWF10], [PTY10], [TDNL07], and virtual worlds [SICH+14], to non-geometry aspects such as story-telling and drama management [NAM06], camera movements [KBUF14] and player [TDNL06] / artificial intelligence modeling [ATAPvL11], [HC04].

From a historical point of view, although procedural methods arose in the field of early video game development, their use was not common in the special effects and feature animation community.

While computer-generated content made first appearance in movies in the late 70s, it was not until the 90s, most notably with the movies Terminator 2 and Jurassic Park, that the film industry started using 3D computer graphics for content authoring and animation. Nowadays, procedural effects for 3D games are similar to procedural effects for movie productions. Differences can be found in the degree of visual fidelity, but these are mainly caused by the real-time demands of 3D games.

3.4 Generative Modeling Language (GML)

Sven Havemann proposes a stack based language written in C++ called Generative Modeling Language (GML), which allows, but is not limited to, creating polygonal meshes [Hav05]. The postfix notation of the language is very similar to that of Adobe Postscript. A stack is used to exchange parameters between functions. The GML interpreter has to ensure that results from previous operators or function calls become input parameters of subsequent functions. Thus, high-level shape operators are created from low-level shape functionality.

3.4.1 Language Elements

GML code is interpreted as a stream of tokens, which can either be a literal, or executable. In case of it being a literal, it is pushed onto the stack. An executable is executed using the stack to provide input.

A literal can be one of the atomic data types: integer values (`42`), floating point values (`13.37`), strings (`"string"`), vectors (`(1.2,3.4)`, or `(1.2,3.4,5.6)`), markers (`[`, {, }) and literal names (`/name`). The type set can be extended on C++ level to create additional literals. Operators (`pop`) and path names (`.name`), for example, are always executable.

Operators An operator pops its inputs from the stack, processes them, and pushes the results back on the stack. It consumes, respectively produces an arbitrary number of input and output elements. Apart from GML's core operators (similar to *Adobe Postscript*), new operators are organized in libraries (dictionaries).

Dictionaries and scoping The concept of a dictionary resembles a list of key/value pairs with the key being a literal name, and the value being any type of token. In the GML, a stack of dictionaries – the dictionary stack – is used for look-up purposes. Flexible scoping is possible, because the topmost dictionary on the dictionary stack that contains a requested name as a key also defines its value. The dictionary stack always contains the *global dictionary* for looking-up the atomic operators. Dictionaries can be pushed onto, and popped from the dictionary stack with the operators `begin` and `end`. Navigation in dictionary hierarchies is performed with path names. The dot prefix of a path name (`.name`) triggers its execution – a dictionary is popped from the stack and the token `name` is pushed back onto the stack.

Arrays and Functions Arrays in the GML can either be literals, or executables. When the operator `]` is interpreted, all elements on the stack until the marker `[` are consumed to form an array. Thus, arrays can consist of elements of different types (also nested arrays are possible). In contrast to literal arrays, the markers { and } are used to create executable arrays (or functions). While literal arrays are pushed onto the stack, executable arrays are executed token by token. The { marker enables the interpreter to treat all following tokens as literals, and to push them back onto the stack. In case of operators, their respective executable name is pushed. The enclosing } marker ends this interpreter mode and pushes the resulting executable array back onto the stack. Executable arrays can therefore be used as functions taking inputs from the stack.

Registers Due to difficulties in keeping track of the stack, the GML offers a convenient, yet efficient way to store values for later use. The main purpose of registers is to have fast local variables, that are valid only in the scope, they are defined in. The usage of registers can be enabled with the operator `usereg`. Values are stored within a register by using `!` followed by a name. Retrieving values is possible with `;` or `:` followed by the name. The difference between `;` and `:` is that by using `;`, the value is just pushed onto the stack. When using `:`, the token stored within the register is also executed. It is good practice to begin a function by popping all needed parameters into registers.

Flow control Looping and conditional branching are imperative features of programming languages. The GML-way of implementing these features is to use special flow control operators (see Table 3.1). While the `map` operator pushes a return value (an array) back onto the stack, any function can leave values on the stack.

3.4.2 Shape Modeling

In the GML, three-dimensional shapes can be generated by means of Euler Operators. They are topological operators that modify incidence relationships and are not related to geometric operations. Mesh manipulation and navigation is also possible on a half-edge level since Euler Operators are not suitable as end-user interface. A powerful mesh data structure is the foundation for the modeling operations. It allows interactive mesh modeling and supports semantic

Flow control	Description
`flag func if`	`func` is executed if `flag` $\neq 0$
`flag func1 func2 ifelse`	`func1` is executed if `flag` $\neq 0$, `func2` otherwise
`func loop`	continues to execute `func` until the `exit` operator is called from within `func`
`num func repeat`	executes `func` `num` times
`init inc to func for`	starting at i = 0, the values `init` + i × `inc` are pushed onto the stack until these values are < `to` (for `inc` > 0), or > `to` (for `increment` < 0); `func` is executed each time
`array func forall`	pushes each element of `array` onto the stack and executes `func` each time
`array func map`	pushes each element of `array`, executes `func`, and pops the element; the resulting elements form a new array

Table 3.1: Conditional branches and loops are implemented in the GML as operators. The token representing the function does not need to be an executable array – it can be any kind of token.

level-of-detail. The GML serves as a platform for a number of applications, because it is extensible and comes with an integrated visualization engine.

The GML uses so-called combined B-Reps as its low-level shape representation. It is a combination of a half-edge datastructure (see Section 2.4.2) with Catmull-Clark subdivision surfaces (see Section 2.4.3). The flexibility of composing models of free-form and polygonal parts is achieved by attaching a sharpness flag to every edge in the mesh, to distinguish between sharp edges and smooth edges. Regions with sharp edges are rendered using polygons, while regions with smooth edges are regarded as a control mesh for subdivision surfaces.

Euler Operators are used at the very core of GML's modeling functionality. Their theoretical foundations are discussed in the work of Sven Havemann. In the GML, five Euler operators (along with their respective inverse ones) are used:

- **makeVFS** (**killVFS**): This operator is used to initially create a shell (connected component) with one face and one vertex. The inverse operator is used to delete a shell together with its last vertex and face.

- **makeEV** (**killEV**): A single vertex and an edge are created by this operator. It splits an already existing vertex into two, and creates a connection (edge) between the two new vertices. The inverse operator removes an edge and merges the two vertices.

- **makeEF** (**killEF**): This operator is used to split a face into two faces, introducing an edge. The inverse operator joins two faces into one.

- **makeEkillR** (**killEmakeR**): This operator is used to replace an edge-ring with an edge. The inverse operator creates a ring within a face.

- **makeFkillRH** (**killFmakeRH**): This operator enables topological changes of an object. By using this operator, a face is created out of a ring-hole. The inverse operator takes a face and turns it into a ring of another face.

While these operators form a complete set of modeling operations, it is a tedious task to directly work with them to create a shape. Thus, the GML offers built-in high-level shape operators that allow a more convenient way of modeling.

Throughout its many years of existence, many generative models have been created using the GML. Figure 3.9 shows two examples from the automotive sector.

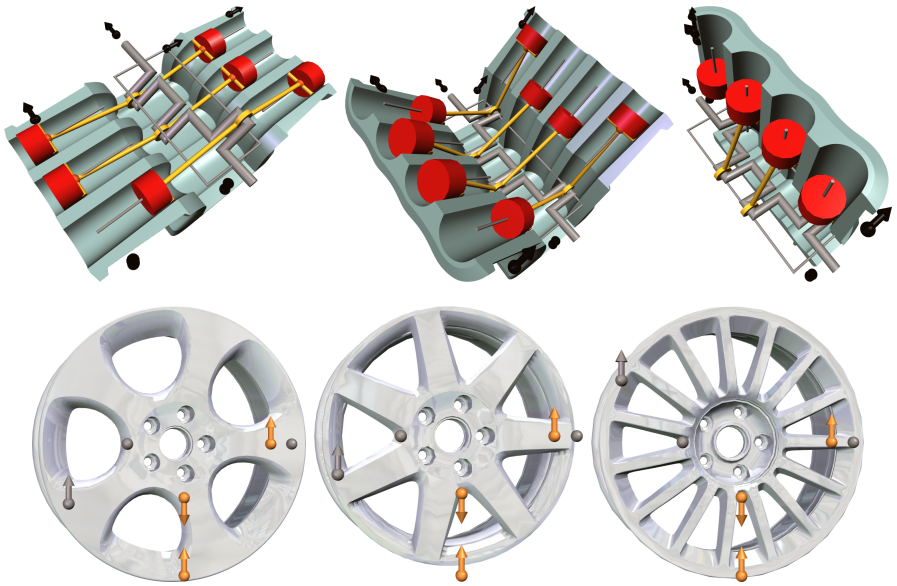


Figure 3.9: This figure shows two examples of generative models created using the GML. The top row shows different configurations of an engine that can be changed by interacting with the black gizmos. Different designs of rims are depicted in the bottom row. Using the gray and orange gizmos, it is possible to cycle through the different designs and further customize diameters and style elements.

The top row schematically depicts the functional principles of different engine types. Several parameters of the engine can be changed by interacting with the black gizmos. It is possible to change the number of cylinders, the angle between cylinder banks in V-engines and the position of the cylinders. By choosing an angle of zero degree, the V-engine becomes a straight engine. The bottom row shows different designs for rims. It is possible to cycle through different designs and to further customize diameters and style elements using the gray and orange gizmos.

The following section presents a use case that demonstrates the effectiveness of generative modeling for mass customization of consumer products using wedding rings.

3.4.3 Application: Wedding Rings

The wedding day is one of the most important days in most people's lives. Many couples choose a wedding ring to dignify this special event (see Figure 3.11). Since a wedding is tailored towards a couple, there is an inherent demand to customize their wedding ring as well. Customization is easily possible when products are made by hand, but due to modern production facilities allowing parameters to be varied with every single produced item, it is nowadays also feasible for automated production. This is the basis for mass customization where designers no longer create a static product (a *design*), but a whole product family (a *metadesign*).

While a metadesign is obtained by generative modeling techniques (see Section 3.3), various ways exist for realizing designs for wedding rings: There are flexible solutions like 3D Modeling Software (see Section 3.1.1), but also specialized jewelry and ring design software:

- *Firestorm CAD* offers the ability to create solid jewelry designs by extruding paths [Bab17]. These paths can be obtained by tracing images, or projecting images onto different surfaces. A large number of diamond layouts are available for use.
- The tools *CounterSketch* and *Matrix* allow designers to customize predefined styles, or even create customizable styles for jewelry [Hig17]. Freeform designs can be created using T-Splines and a large array of tools for laying out and adjusting gems.
- *JewelCAD* is a specialized 3D freeform modeler with modeling tools that allow much freedom in creating artistic and stylish designs [Li17]. It facilitates the import of profile paths from 2D sketches for path extrusions defining patterns and lattices.
- *3Design* combines a library of design elements with freeform modeling of artistic jewelry designs [Guy17]. Tracing images to create paths for extraction enables an easy workflow from 2D to 3D.

In the article "Ring's Anatomy – Parametric Design of Wedding Rings" [BSK+12] by René Berndt et al., we present an encoding of the metadesign of weddings rings from the design space of the Ringmanufaktur JohannKaiser[2] in the GML.

Ring features

To create a metadesign, the features of the product family have to be identified. An obvious feature of wedding rings, or rings in general, is to fit on human fingers – thus they resemble the shape of a circle on the inside. There are various other distinct features. The profile is a very characteristic one defining the shape of the ring. Some common profiles are shown in Figure 3.10.

The flat section profile is the traditional wedding ring profile, which is flat on the inside and on the outside. D-shaped profile rings have a flat profile on the inside and a bent profile on the outside. Rings with halo profile have a perfectly round cross section. Oxford court is an oval profile with flat rounded internal and external facets, typically resembling the shape of an ellipse.

[2] http://www.johannkaiser.de

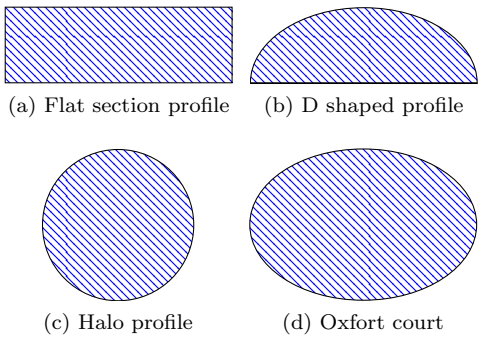


Figure 3.10: A ring's profile is a very characteristic feature. Common ring profiles are: flat section profile (a), D-shaped profile (b), halo profile (c) and Oxford court (d).

Another characteristic property of a wedding ring is the material it is made of. Various metals like gold, silver, platinum, but also palladium, stainless steel, or titanium are typically used for wedding rings. Also a combination of different materials is possible. The surface structure of the material can be varied from polished, highly specular surfaces, over glossy, to different kinds of abraded ("brushed") surfaces. Like with different material combinations, also combinations of different surface structures are possible (and rather common).

The surface of a wedding ring can be further garnished using engravings and gemstones. Engravings include artistic strokes and patterns, but also simple text, typically the names of the couple and the date of the wedding on the inside. By applying gemstones, a wedding ring's character becomes even more noble. Typical gemstones used are diamonds, sapphires, rubies, emeralds, amethysts and aquamarines. While gems also add value, their distribution pattern often also has a symbolic meaning.

A last feature of wedding rings is directly related to the placement of gemstones. In order for gemstones to be kept in place, a fixed connection with the ring has to be established in the form of a socket. These sockets have to be incorporated into the surface of the ring.

Some of the aforementioned properties can be seen on the wedding rings in Figure 3.11.

Parameterization

Following the observations made regarding a ring's features, René Berndt used the following parameters to create a metadesign for the basic shape of wedding rings:

- a non-self-intersecting profile polygon,
- the angular step size defined by the number of supporting profiles to be placed around a ring's center,
- the radius,
- a vertex transformation function.

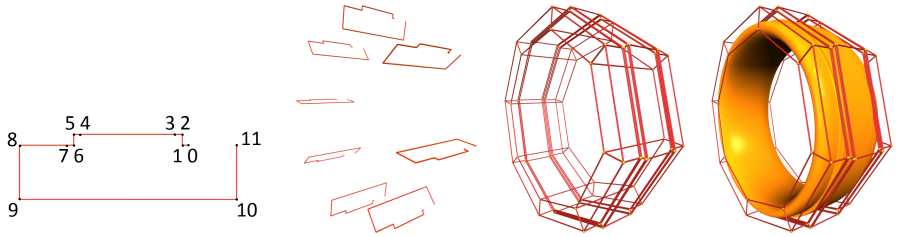
Figure 3.11: Apart from noble materials (here: platinum), the surface of a wedding ring can be garnished with material finishes (here: "brushed" surface), engravings (here: names and dates), as well as gemstones (here: diamonds).

Further design variations can be created by applying a set of transformation functions (e.g., sine transformations) to vertices of the profile polygons. These transformation functions can be combined to cover the variations of a design space.

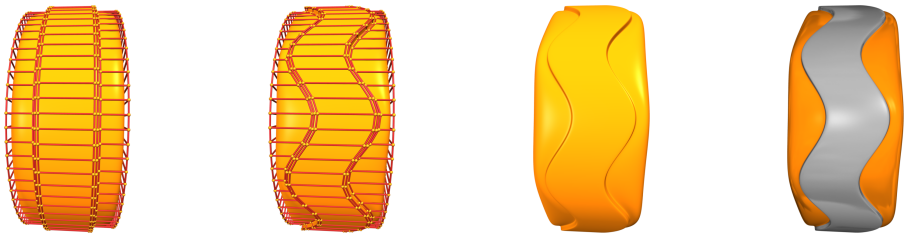
Figure 3.12 shows the different stages of creating the basic shape of a wedding ring. Starting with a profile polygon, a number of copies are placed radially around the origin. These polygons are connected using the `bridgerings` operator to form the control mesh for a subdivision surface (see Section 2.4.3). It expects two halfedges of corresponding vertices of two polygons (faces).
The initial control polygon defining the ring's surface is defined by a function taking the profile polygon and the rotation angle as parameters. In order to further customize the ring, an additional set of parameters are needed. These custom parameters depend on the particular transformation used – a sine transformation, for example, requires three parameters: frequency, amplitude, and the indices of the points of the profile polygon to which the transformation is to be applied to. The transformation function can transform the profile points arbitrarily.

Materials and Engravings

After creating the basic shape of a ring, materials and engravings are added using different techniques. Depending on their impact on the ring's geometry, we separate engravings (the so-called meso-structure) from the material and its surface structure (the micro-structure). Engravings are defined and applied as (per-vertex) displacement maps, with the idea to create a micro-tessellation of the surface only in regions with displacement. Material and surface structure



(a) the profile polygon (b) radial placement around center (c) connecting profiles (d) subdivision surface



(e) identity transformation (f) sine transformation (g) decorative wave (h) different coloring

Figure 3.12: The basic shape of most wedding rings can be constructed in a few stages: A number of copies of the profile polygon (a) are placed radially around the center (b). Consecutive profiles are connected to form a control mesh (c) to define a subdivision surface forming the actual ring shape (d). Exchanging the default identity profile transformation function (e) to a selective sine transformation (f) creates a decorative wave on the surface (g). In a post-processing step the wave faces can be colored differently.

are regarded as micro-structure. They are applied as a combination of material (consisting of: ambient, diffuse, and specular color, illumination and shininess) and normal maps for different surface structures.

Assigning materials to parts of the ring is done in a few lines of code:

```
1  :edges
2  [ 3 ]
3  /silver
4  RING.Tools.colorize
```

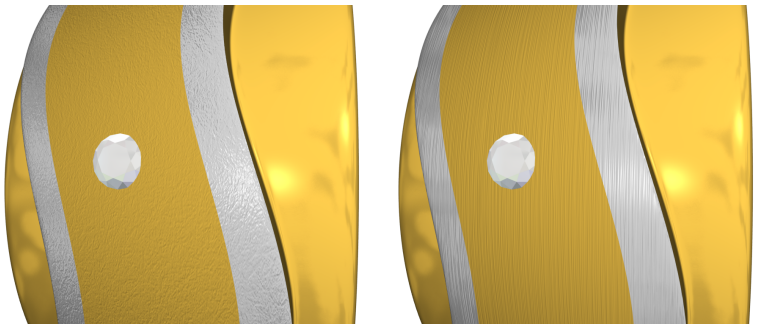
The parameters for `RING.Tools.colorize` are an array of half-edges (line 1), the indices of the affected faces, and the material name. Figure 3.12 (h) shows the result of applying this code snippet. The material is applied in the final rendering stage using the standard Blinn-Phong shading model combined with an approximation of the Fresnel reflection term that can model anisotropic highlights. To make the surface look metallic, the specular highlight is modified by the color of the material. We use a static cube map to reflect the environment.

The material's surface structure can be controlled by applying several predefined structures, like "brushed", or "sand" to the surface (see Figure 3.13).

Displacement Mapping

The basic idea of displacement mapping is to take sample points from a surface, and displace in the direction of the surface normal in a distance corresponding to the gray value in a supplied height map. The normal vector of the displaced vertices is calculated by combining the surface normal with the supplied normal map.

Displacement mapping techniques can generally be classified in per-vertex and per-pixel techniques. Some techniques only affect surface normals and, thus, the lighting calculation – points are virtually displaced corresponding to texel centers. Consequently, per-pixel displacement is carried out by the texture unit of the fragment shader. Per-vertex displacement changes the positions of the of the original mesh vertices, and is therefore implemented in the vertex or geometry shader. So per-vertex displacement mapping creates actual geometry, whereas per-pixel displacement only creates the visual effect of a displacement. László Szirmay-Kalos and Tamás Umenhoffer give an overview of current displacement mapping techniques [SKU08].



(a) surface structure: "ice" (b) surface structure: "brushed, coarse"

Figure 3.13: The operator `nmp-material` creates different surface structures, for example: "ice" (a), or "brushed, coarse" (b).

In order to achieve the desired effects, the `nmp-material` operator relies on predefined normal maps. The parameters for `nmp-material` are a material id, a starting half-edge, an extend (in two dimensions), as well as scaling.

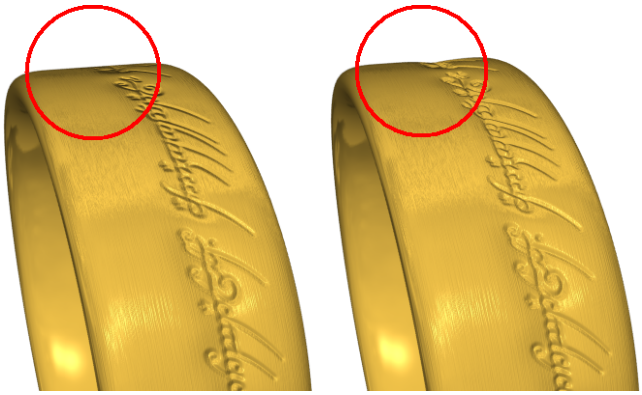
The material id defines the type of surface structure – Table 3.2 gives an overview of all currently available surface structures.
Engravings can be created in a few lines of GML code using the operator `dsp-displacetexture`. The operator's parameters are the file names of displacement and normal maps, a scale factor, a starting half-edge, and an extent.

The rationale behind using per-vertex displacement mapping is that the metadesign is not only used for visualization, but also for rapid-prototyping purposes – the per-vertex approach allows exporting of displaced geometry. While per-vertex displacement requires a rather dense tessellation for detailed displacement maps, a view-dependent tessellation scheme overcomes this issue. Depending on the distance between camera and geometry, as well as on ren-

Material id	Material description
0	ice
1	brushed, 45 degree, coarse
2	brushed, 45 degree, fine
3	brushed, coarse
4	brushed, fine
5	sand, coarse
6	sand, fine
7	brushed, concentric

Table 3.2: A total of eight different surface structures are currently available via the `nmp-material` operator in the GML.

dering speed, several levels of subdivision are available. Another advantage of per-vertex approaches is a correct visualization at the silhouette. The rings in Figure 3.14 illustrate the visual superiority of per-vertex displacement mapping.



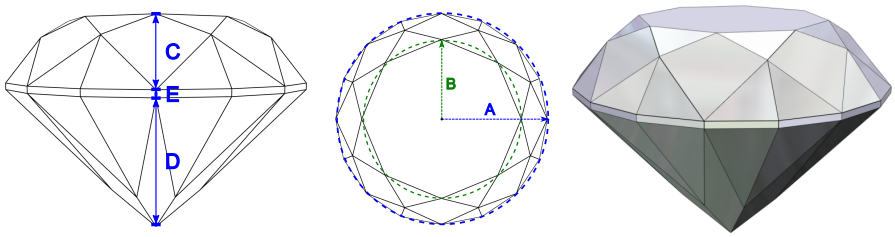
(a) normal mapping (b) displacement mapping

Figure 3.14: The comparison of normal mapping (a) and displacement mapping (b) reveals visual deficiencies at the silhouette as well as a lack of "depth" when using normal mapping.

Gemstones

Wedding rings are often decorated with gemstones to emphasize their character. Our metadesign features gemstones of the round brilliant cut type, due to its dominant position in the market [HRJS98]. Similar to Scott T. Hemphill et. al, Ulrich Krispel's mathematical model consists of a convex polyhedron, a 3D convex polytope (see Section 2.5.2), representing the surface of the gemstone. A convex polyhedron is defined by the planes of its facet – three 3D points. The orientation of these points is important and allows to distinguish between below and above the plane.

Figure 3.15 (a and b) shows the five parameters used to define the shape of a diamond: total diameter, table diameter, crown height, pavilion height and girdle height.



(a) side view (b) top view (c) rendering

Figure 3.15: Five parameters are used to define the shape of a diamond. The parameters are total diameter (A), table diameter (B), crown height (C), pavilion height (D) and girdle height (E).

The system differentiates between the instantiation of a diamond, given a parameter set, and the placements of instances of this diamond. The evaluation of the convex polyhedron in the GML creates the instance geometry of a gemstone as can be seen in Figure 3.15 (c). Gemstone instances can be placed by specifying the apex point, an up-vector, and a scale factor. The apex point is typically determined by a ray-mesh intersection with the ring's surface.

Sockets

Sockets for gemstones, or other sharp features, are insufficiently handled by displacement mapping techniques. A typical circular socket for a single gemstone requires a very fine tessellation in order to create a visually appealing result. Through the use of Constructive Solid Geometry (see Section 2.5.3), complex, sharp surface features can be realized.

With my help to integrate this feature into the system, René Zmugg realized a image-space method for CSG. Since this method is limited to image-space only, it does not alter a ring's geometry. Therefore, it can not be used for rapid prototyping, only for visualization purposes. The image-space method relies on knowledge about the depth complexity of the *plus-object* (the ring) and the *minus-object* (e.g., a cylinder). Figure 3.16 depicts the depth complexity as the maximum number of times an arbitrary ray can intersect with a given object.

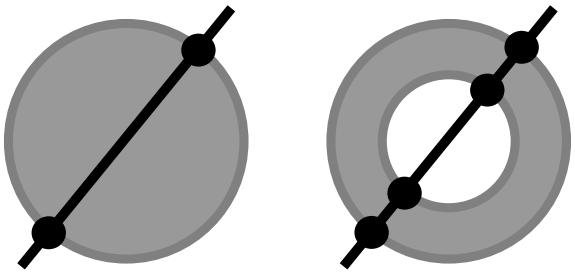
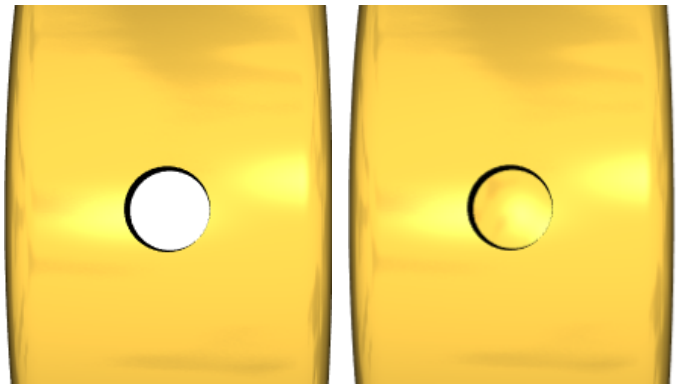


Figure 3.16: The left disc has a depth complexity of two, as an arbitrary ray intersects the disc at a maximum of two points. Arbitrary rays can intersect a ring at two points too, but at a maximum of four points (right).

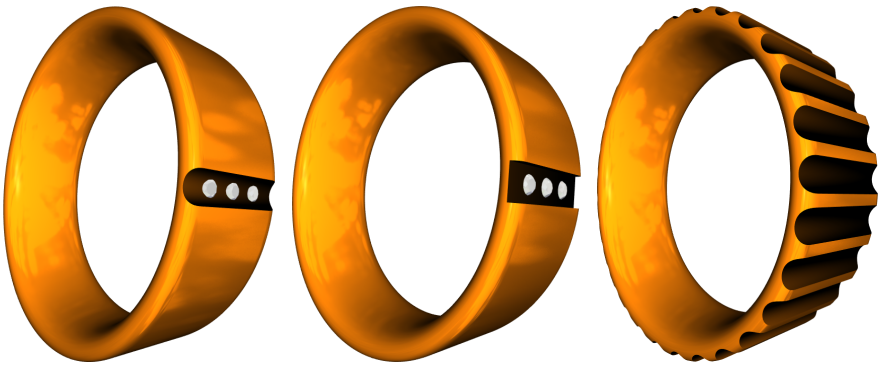
The algorithm proceeds by creating the specified amount of depth-peels for the two objects and combines the necessary parts to create the final image. If the amount of depth-peels is chosen too low it might not be possible to see the back part of a ring through a subtracted cylinder as illustrated in Figure 3.17.



(a) two depth-peels (b) four depth-peels

Figure 3.17: Using the correct number of depth-peels shows the back part of the ring as visible through a hole cut into the front part (b). With the number of depth-peels set too low, the back part of the ring is not visible (a).

Too many depth-peels do not affect the visible result, but affect the execution time. Examples for rings created through the use of Boolean operations with cylinders and boxes are shown in Figure 3.18.



(a) round cut (b) square cut (c) multiple round cuts

Figure 3.18: Rings with round (a), square (b) and multiple (c) cuts can be created. Boolean operations using image-space CSG greatly enhance the possibilities of the metadesign.

Examples

With all components at hand, our system is capable of generating a variety of ring designs (see Figure 3.19). The implementation of a the metadesign took about six weeks, with a resulting GML code size of about 100 KB. JohannKaiser are using this system for their wedding ring configurator *REx*.

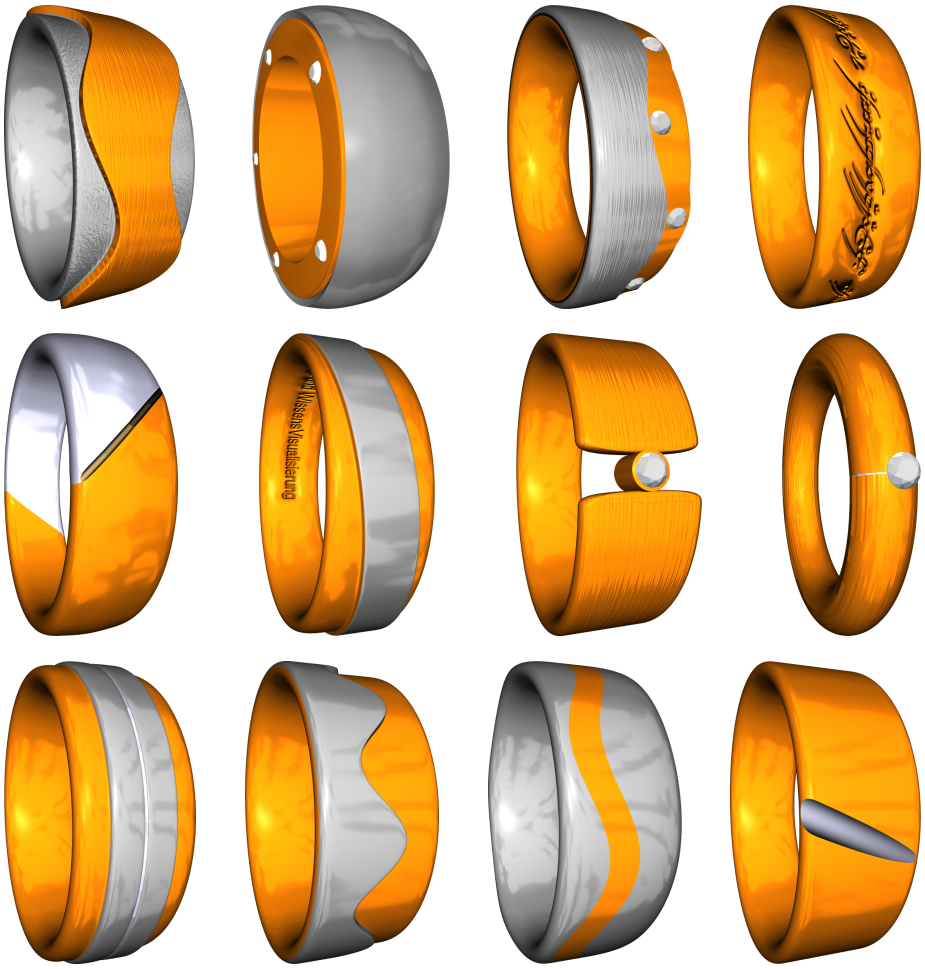


Figure 3.19: The output of the parameterization process is a metadesign that can generate a variety of ring designs.

3.4.4 Application: Serverside Rendering for Generative Content

The presented metadesign of wedding rings in Section 3.4.3 efficiently encodes the whole product family in about 100 KB GML code. Apart from textures (including normal and displacement maps), this solution is ideal for low-bandwidth applications. However, due to greater demands on the graphics hardware (for per-vertex displacement mapping and image-space CSG), it is not suitable for low-cost client computers, or tablets and smartphones. Supporting the variety of different platforms (especially mobile platforms) would also require a large amount of development work for a C++ application, like the GML. Additionally, there is the question of intellectual property protection (IPP), in the case of the wedding rings.

The problem of supporting different platforms can be solved by delivering the visualization through a web browser. René Berndt, Dieter W. Fellner and Sven Havemann describe a browser plug-in based approach that (nowadays) has several disadvantages [BFH05]:

- It is not possible to protect the IP of a generative description, since everything is processed by the plug-in and therefore has to be transferred to the client.
- There is a significant inhibition threshold to install a plug-in, if even possible on the target platform.
- The quality of the visualization heavily depends on the capabilities of the client's hardware and thus cannot be guaranteed.
- Maintenance and deployment of a plug-in are time- and labor-intensive and often cannot be financed over a longer period.

A possible step could be the implementation of GML using JavaScript and WebGL [Khr14] technologies. While the problem of IPP would still persist, it would require a rebuild of the GML ecosystem in a web browser – which is a time-consuming task without the help of tools like Emscripten [Zak11].

A main advantage of the browser-only approaches is the elimination of server-side infrastructure to the greatest possible extent since only the generative descriptions need to be transferred to the client.

In the article "A Scalable Rendering Framework for Generative 3D Content" [SBEF14] by Christoph Schinko et al., we present a hybrid solution to overcome the IPP problem. In our scenario, server-side hardware is used to allow the rendering to be shifted away from the client. The client uses X3DOM [BEJZ09] technology to display a combination of simple proxy geometry and rendered images. This way the interactivity of navigation is preserved, the rendering quality is independent of the client's hardware and the IPP problem is solved. A drawback is the need of powerful server hardware, but due to the business model for this application being tailored towards jewelers and wedding ring studios, the amount of customers is manageable.

Hybrid Rendering

Hybrid rendering approaches create images by combining parts generated both on the client and on the server. The workload is typically shifted between the components to achieve different goals, like for example: a maximum of rendering performance or a minimum of data transferred.

RealityServer from Migenius [Mig17] is a software product for producing photo-realistic renderings of 3D models. A server component takes care of producing rendered images from a given 3D model using NVidia *Iray* technology [Nvi17b]. The images are streamed to the client without the requirement of plug-ins. Furthermore, cloud computing enables the server component to scale with regards to rendering performance and concurrent users.

Scientific visualization tools like *ParaView* from Kitware [Kit17a] are suited for remote visualization, as described by James Ahrens, Berk Geveci and Charles Law [AGL05] and Andy Cedilnik et al. [CGM+06]. Because of its component-based structure, data processing and data rendering can be done on the same machine or can be run separately on different computational resources. Thus, a client can receive either rendered images coming from the rendering resource or pre-processed 3D content. *ParaView* uses the *Visualization Toolkit* [Kit17b] as the data processing and rendering engine.

Interactive server-side rendering approaches are also successfully applied to cloud gaming. Popular services like *Playstation Now* from Sony [Son17] and *Geforce Now* from NVidia [Nvi17a] offer cloud-based gaming platforms. The game application runs on a remote server and can be accessed over the internet by various devices. The user's input is sent to the server and then passed to the game application. The output is encoded as a video stream and sent to the client.

The *webVis/instant3Dhub* platform from Fraunhofer IGD [Beh17] combines a web-components based framework (*webVis*) with a visual computing as a service infrastructure (*instant3Dhub*). By combining the two components, a powerful and comprehensive solution for interactive 3D visualization is available. The system adapts to a given environment by providing client, server and hybrid visualization techniques to optimize the delivery of complex data sets. Furthermore, it provides services like data transcoding and mesh optimization.

System Architecture

The system architecture consists of four different layers (as can be seen in Figure 3.20):

- **Client layer** The client layer represents the web application within a browser, which runs on a laptop, desktop, tablet or mobile device.
- **Application layer** The application layer represents all information and constraints of the domain. In the context of wedding rings, it contains the metadesign.

- **Hardware Accelerated SErver-side Rendering (cHASER) layer** The cHASER layer is responsible for session management and distribution of the workload to the available GML Rendering Units (GRU).
- **GRU layer** A GML Rendering Unit encapsulates a single GML interpreter to create a rendering. The GRUs can either be located on the same server, which is running the cHASER or distributed on an arbitrary number of machines.

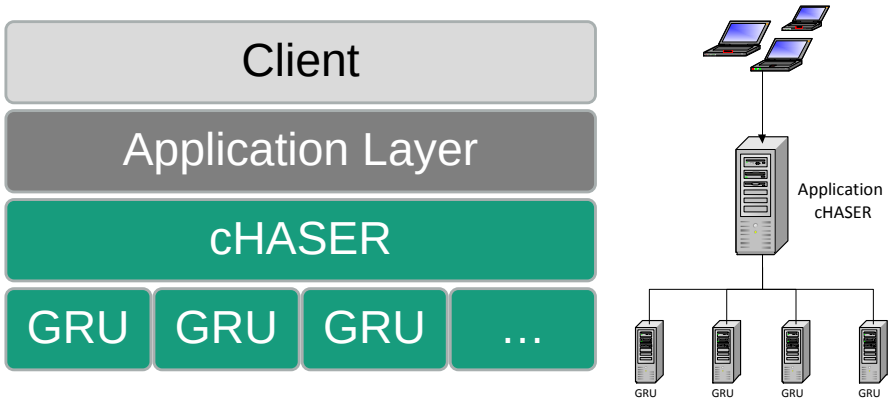


Figure 3.20: The system architecture consist of four layers. Clients are communicating with an application layer representing the generative domain. The main component of the architecture is the cHASER layer for controlling the workload for the GRUs, which are responsible for rendering.

The interfaces between client and cHASER layer, as well as cHASER and GRU layer are implemented as RESTful web services [FT00].

GML Rendering Unit (GRU) The GML Rendering Unit exposes a GML interpreter using a small RESTful API. Table 3.3 shows the functions to execute GML code, request a rendering, and restart a specified GRU.

Resource path	Method	Description
/GRU/{id}/interpreter	POST	Executes the provided GML code on the specified GRU.
	GET	Requests a rendering from the specified GRU.
	DELETE	Restarts the specified GRU.

Table 3.3: The RESTful API of a GML Rendering Unit (GRU) is small due to the possibility to post and execute GML code on the remote interpreter.

Potential harm to a GRU due to the execution of arbitrary GML code is prevented by the cHASER layer. Direct exposure of the GRUs to a client is not possible.

Hardware Accelerated SErver-side Rendering (cHASER) As a central part of the system, the cHASER layer is responsible for the session management, for delivering renderings through the cHASER cache, as well as for distributing the incoming requests to the available GRUs via a GRU dispatcher.

The GRU dispatcher is responsible for distributing the requests from a session to the available GRUs. While a GRU initially gets assigned to a session, it is not exclusively occupied (e.g., after a certain timeout period, a GRU is free to be used by other sessions). In case the application layer requests the execution of GML code for a given session, the GRU dispatcher checks, whether the GRU has not been used otherwise. If so, the operation can be executed on this GRU. Otherwise, a so-called "GRU fault" happens and the dispatcher looks for the GRU, which has been idle for the longest time and attaches it to that session. The state of the session has to be restored to the GRU by replaying all GML commands and restoring the latest versions of the modelview- and projection matrix. This task can be rather time consuming, but should not happen too often, since the typical use case after initially requesting a rendering are consecutive requests to changing the camera.

The cHASER cache is used by the GRUs to store renderings – a file-watcher synchronizes the GRUs with the cHASER layer. For delivery, renderings are not sent through the RESTful interface, but are accessed directly by the clients. Table 3.4 gives an overview of the available functions of the RESTful API.

Resource path	**Method**	**Description**
/api/session	POST	Creates a new GML session and returns the corresponding session id.
/api/{sId}	POST	Executes the provided GML code on the assigned GRU.
	DELETE	Ends the GML session.
/api/{sId}/rendering	GET	Renders the given GML session to an image and returns the URL to the image.
/api/{sId}/x3dom	GET	Requests an X3DOM representation and returns the URL to the model.
/api/{sId}/modelviewmatrix	PUT	Sets the modelview matrix for the given session.
/api/{sId}/projectionmatrix	PUT	Sets the projection matrix for the given session.
/api/{sId}/rendering/size	PUT	Sets the size of the rendering.

Table 3.4: The RESTful API of the cHASER layer covers all requests from the application layer regarding session management and rendering. Please note that *sId* is short for *sessionId*.

The functions are described in detail:

- **Start session (POST)** When stating a session, the session management

allocates and returns a Universally Unique IDentifier (UUID), which is used by all subsequent API calls. The session stores the history of all GML calls in order to restore the state for a GRU.

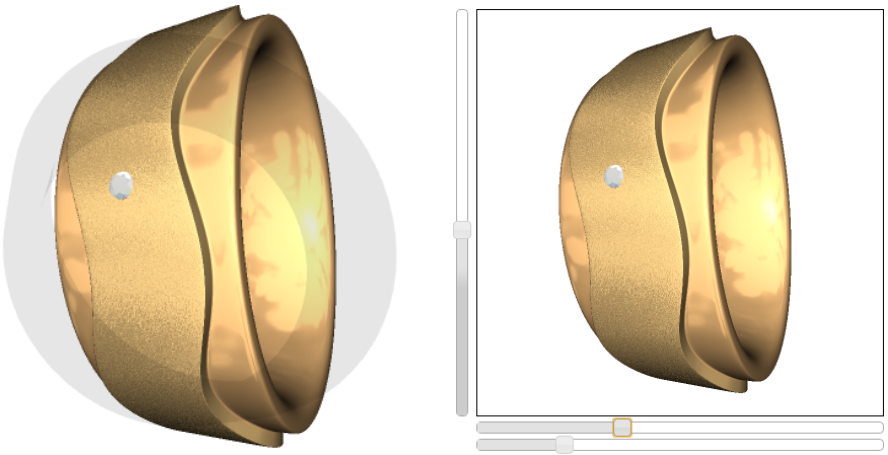
- **Execute GML code (POST)** The GML code to be executed is sent to the assigned GRU.
- **End session (DELETE)** The session is terminated and the respective renderings in the cHASER Cache are removed.
- **Request rendering (GET)** If the client requests a rendering, the request is forwarded to its attached GRU. The response is sent when the rendering is complete and contains a URI for the rendered image served from the cHASER cache.
- **Request X3DOM (GET)** If the client requests the X3DOM representation of the model (a low-polygon model), the request is forwarded to its attached GRU. The response is sent when the representation is created and contains a URI for the model served from the cHASER cache.
- **Setting modelview-/projectionmatrix (PUT)** In order to keep the view of the X3DOM proxy and the parametric model in sync, modelview and projection matrices can be set. The current view of the X3DOM representation is applied to the GRU for the next rendering request.
- **Setting rendering size (PUT)** The size of the next rendering can be controlled by setting its dimensions.

Client-side Visualization and Navigation

Visualization and navigation on the client are done in a web browser using X3DOM Technology [BEJZ09]. For the visualization proxy geometry for direct 3D manipulation of the scene is displayed in combination with rendered images when no scene manipulation is happening.

A typical rendering task consists of creating a new session on the cHASER layer and selecting (or parameterizing) a GML model for rendering. Proxy geometry is requested and sent to the client to provide the user with convenient and immediate manipulation feedback, see Figure 3.21 (a). The requested proxy geometry consists of a low-polygon version of the requested GML model. Initially, the proxy geometry is not shown to the user; it is set to be transparent. The next step is to request an initial rendering: modelview and projection matrices are passed to the cHASER layer and the corresponding rendered image is sent to the client. A mouse-click on the X3DOM canvas (i.e., on the rendered image) activates (displays) the navigation proxy geometry. During mouse interaction the proxy geometry is directly manipulated and stays visible until the mouse button is released. Then, the proxy geometry is deactivated (by setting it to be transparent) and a new rendering is requested.

In case the client does not support X3DOM (e.g., due to lacking browser support or insufficient hardware) a fallback solution is provided. Since it is not possible to display proxy geometry, navigation is achieved by using three sliders for pitch, roll and zoom as can be seen in Figure 3.21 (b). In this mode, renderings are requested permanently to supply the needed feedback for the user.



(a) proxy geometry (b) fallback with sliders

Figure 3.21: Different modes of interaction with the geometry are available. A proxy object is used to directly manipulate the object in the X3DOM canvas (a). As a fallback solution (for browsers not supported by X3DOM) sliders can be used for zooming, translating and rotating (b).

Performance and Evaluation

In order to find a trade-off between the number of clients concurrently handled by a server and the rendering time needed, the performance of the system has been evaluated. On the one hand, rendering requests from clients need to be processed as fast as possible to minimize waiting time, since it is unsuitable for a client to wait more than a few seconds to receive a rendering. On the other hand, it is not reasonable to maintain a dedicated server for every client.

Tests with dedicated client and server show that the average rendering time for a predefined model with a different number of threads and GRUs increases almost linearly. The delivery time, from a server point of view, shows that a single GRU is not able to produce as much throughput as two or more GRUs. When testing an identical number of clients and GRUs (n clients for n GRUs), an almost linear increase in rendering time as well as an almost constant throughput on the server is observable. These observations allow for an easy dimensioning of server hardware, based on an expected number of concurrent clients.

The wedding ring demonstrator shown in Figure 3.22 is providing an interactive high-quality user experience, even on devices without out-of-the-box WebGL support. On WebGL-enabled devices, proxy geometry offers a more intuitive navigation.

3.5 Summary

This chapter focused on primitive modeling, semantic modeling, and generative modeling. The subsequent section was used to introduce the Generative Modeling Language (GML) together with an application: a generative description (a so-called metadesign) of a wedding ring design space for a web-based product

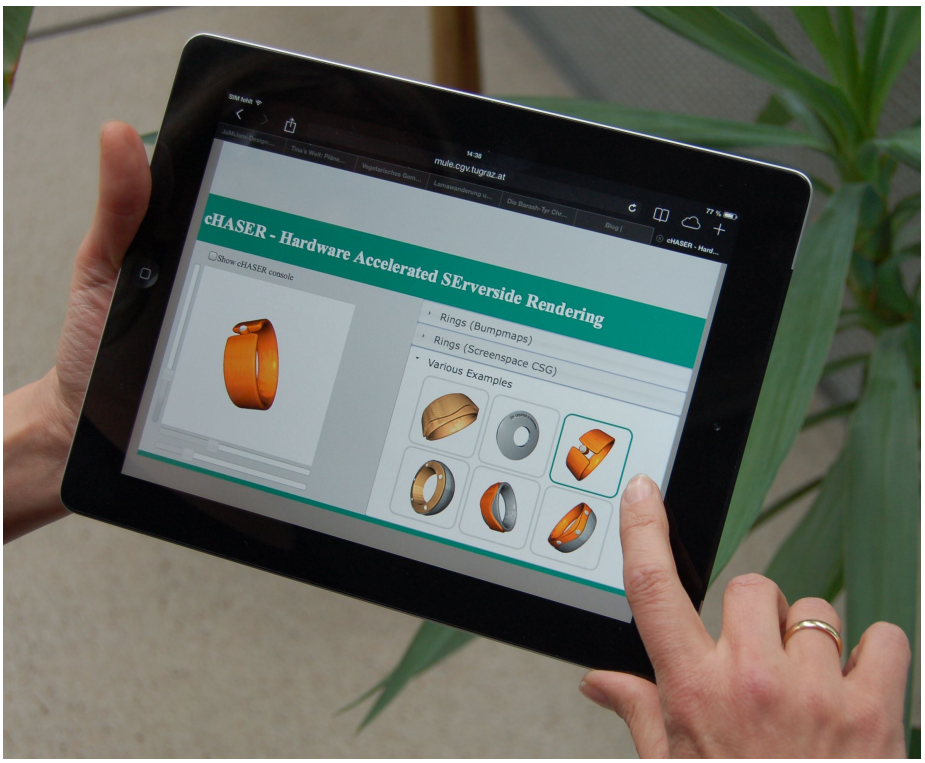


Figure 3.22: Configuring a wedding ring using an Apple iPad 2. The image based approach can still provide an interactive high-quality user experience, even devices without out-of-the-box WebGL support.

configurator. While this application satisfied all requirements for the defined use case, it also demonstrated problems arising from using a platform-specific language on unsupported platforms (a web browser). In the case of the wedding ring configurator, this problem transformed into a feature and served the purpose of intellectual property protection. However, there are many scenarios, where it is necessary to implement and maintain several generative descriptions on different platforms. The next chapter will present a system capable of supporting many different platforms.

Chapter 4

Meta Modeler: Euclides

The results presented in the previous chapter demonstrate the potential of generative modeling techniques. Independent of the beauty that lies within the creation of highly complex shapes based on a set of formal construction rules, it also shows a major drawback. The potentially general approach to use construction rules has to be re-implemented on every single platform. This is problematic in larger systems where different domain-specific languages and tools come into play. The effort to implement and maintain several generative descriptions is significant.

The novel approach presented in the following chapter addresses this problem by introducing the concept of generative meta-modeling. A single implementation of all necessary construction rules in an easy-to-use language can be translated to different target platforms. Even though there is only a limited amount of target platforms available to date, the potential of the approach becomes visible.

Contents

4.1 Overview

The novel meta-modeler approach called *Euclides* is introduced in the article "Procedural Modeling in Theory and Practice" [USF10] by Torsten Ullrich, Christoph Schinko and Dieter W. Fellner.

It is a major problem that the need to use a programming language is a significant inhibition threshold especially for domain experts like archaeologists, architects, designers, etc. who are seldom experts in computer science and programming. The introduction of a new dimension of complexity and additional dependencies by a programming language further contribute to that problem.

Additionally, the problem of file format conversion is not solved with generative modeling techniques. If a shape does not only contain static geometry but algorithmic descriptions, the file format also depends on the languages and the interpreter that is able to execute the script. Our meta modeler approach presented in this chapter represents a new possibility to create procedural models in a beginner-friendly way. Additionally, we address the file format problem by using a consistent intermediate representation serving as a basis for back-end exporters to different languages and different platforms. Thereby, we reduce the dependencies to scripting and rendering engines. Due to its high-level representation of the input code, the level of abstraction can be preserved after translation to target code. The clear correspondence between input and translated code simplifies debugging and reuse.

Potential execution of arbitrary code represents a significant security threat, especially in service-oriented environments. In Euclides this threat can be reduced by security features of the Java virtual machine and by exercising control over the use of libraries and native code. Many security relevant aspects of programming require access to system resources. This functionality can be made available in Euclides via native code of the target platform. By restricting the use of native code to trusted libraries, security risks can be reduced.

4.2 Architecture

When trying to combine different generative modeling approaches, a question arises: Is it possible to achieve a conversion between file formats, respectively languages? The simple answer is: Yes, but only with considerable expenditure. Because of differences in the intended purpose of the languages as well as paradigmatic variations it is a rather difficult task. In order to be able to cover a variety of approaches it would be necessary to implement converters that differ in the source as well as in the target language.

Our meta-modeler solution represents a common ground for generative modeling approaches, thus avoiding the necessity to create converters to and from all languages. It enables the user to create generative models represented by a single language, but allows a variety of output representations (targets) to be generated by following the principles of a transpiler. Figure 4.1 gives an overview of architecture and available targets.

The choice of language fell on JavaScript since it is a beginner-friendly, yet powerful language. JavaScript and its dialects are widely used in applications and on the Internet: in modern web development frameworks, when working with 3D content in the web browser, in the Adobe Creative Suite, in interactive

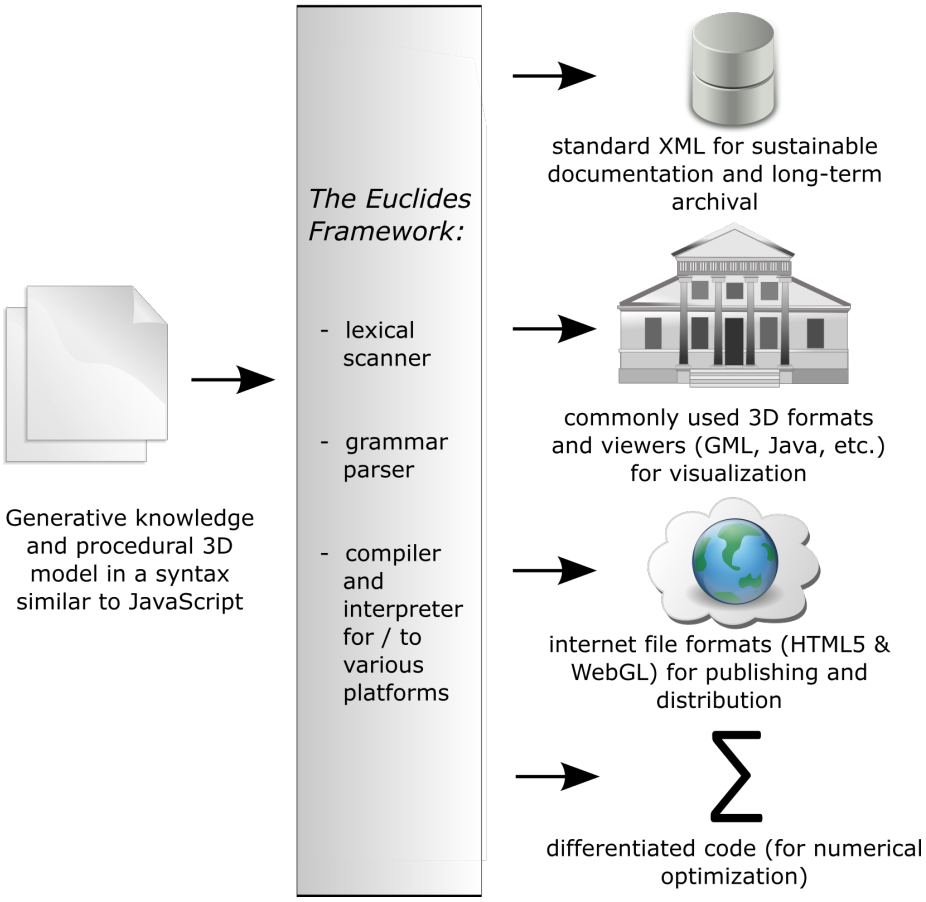


Figure 4.1: The meta modeler approach offers compiler and interpreter for many platforms, a XML target for archival, Java and GML as exemplary 3D target platforms, a HTML/WebGL target, and a target for numerical optimization.

PDF files, in Apple's Dashboard Widgets, in Microsoft's Active Scripting technology, etc. Consequently, there is a large pool available of documentation and tutorials to introduce the language [VV04], [Fla11], [Joh16]. Often needed data structures, algorithms and routines for geometric modeling (such as vectors and matrices) are either available as first class citizens, or as libraries. JavaScript incorporates features like dynamic typing and first-class functions. But the most important feature of JavaScript is: it is already in use by non-computer scientists - namely designers and creative coders. To further simplify the syntax we decided to omit language features not needed for generative modeling like prototypes and classes and add vectors and matrices as first class citizens.

The meta-modeler approach adopted for Euclides differs from other modeling environments in the aspect of target independence. Usually, a generative modeling environment consists of a script interpreter and a 3D rendering engine. A generative model is interpreted directly to generate geometry, which is then visualized by the rendering engine. As described in the article "Modeling Procedural Knowledge – A Generative Modeler for Cultural Heritage" [SSUF10a]

Transpiler

A transpiler is a type of compiler that takes source code written in one programming language as input and creates equivalent source code in the target programming language as output, whereas a compiler typically creates an executable program. Looking at the generated output, differences in the levels of abstraction between a transpiler and a compiler are evident. While a transpiler translates between programming languages operating at roughly the same level of abstraction, a typical compiler creates low-level machine code. Both, transpiler and compiler, initially perform similar tasks, like lexical analysis, parsing and semantic analysis. The creation of an intermediate representation marks the difference. For a compiler, this representation usually is of a lower level of abstraction with respect to the source code as it is for a transpiler. After an optional optimization step, source code in the target programming language is created. Transpilers may either create code in a target programming language with a focus on being as close to the source code as possible (e.g., to ease debugging of the input source code), or focus on different things, like execution speed, or file size. In general, transpilers are also not limited to a single target language.

Bjarne Stroustrup describes an early transpiler called Cfront translating C++ to C (first commercial release in 1985), which is known as the original compiler for C++ [Str07]. In recent years, a focus on increasing brevity and readability of JavaScript produced the programming language CoffeeScript[a]. It is a good example of translated code that corresponds very clearly to the input source code. An effort to ease the development of large applications with JavaScript lead to the birth of the programming language TypeScript[b]. Both languages are transpiled to JavaScript. A different kind of translation is performed by EMScripten, a LLVM to JavaScript compiler [Zak11]. LLVM is a compiler infrastructure project with an intermediate representation for code compiled from languages such as C, C++ or Objective-C. LLVM output used by EMScripten is similar to assembly language – the generated JavaScript code is not meant to be human-readable.

[a] http://coffeescript.org/
[b] http://www.typescriptlang.org/

by Christoph Schinko et al., source code is not interpreted but parsed into an intermediate representation, an abstract syntax tree (AST). After a validation process it is translated into a target language. The process of

$$\text{parsing} \Rightarrow \text{validating} \Rightarrow \text{translating}$$

offers many advantages. The basic steps are mentioned in the article "The Rules Behind – Tutorial on Generative Modeling" [KSU14] by Ulrich Krispel, Christoph Schinko and Torsten Ullrich.
Figures 4.3 and 4.4 outline the compilation process and show the main data structures – especially the AST. First, the input source code is passed to lexer and parser. The sequence of characters is converted into a sequence of tokens by special grammar rules forming the lexical analysis. For instance, some languages

only allow a limited number of characters for an identifier. In Euclides, all characters `A-Z`, `a-z`, digits `0-9` and the underscore _ are allowed with the condition that an identifier must begin with a character, see Figure 4.2.

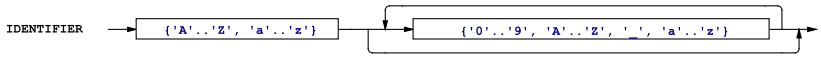


Figure 4.2: The lexer rule *identifier* allows characters `A-Z`, `a-z`, digits `0-9` and the underscore _ with the restriction that it must begin with characters.

These lexer rules are embedded in another set of rules – the parser rules. They are used to analyze the resulting sequence of tokens in order to determine their grammatical structure. A complete grammar consists of a hierarchical structure of rules for analyzing all possible statements and expressions that can be created with a language, thus forming the syntactic analysis.

For each construct of the language a set of rules is validating syntactic correctness. At the same time actions within these rules create the intermediate AST structure representing the input source code, as mentioned in the article "Minimally Invasive Interpreter Construction – How to reuse a compiler to build an interpreter" [SUF12] by Christoph Schinko, Torsten Ullrich and Dieter W. Fellner. Within these actions, an abstract factory, like described in [FFBS04], called `ASTFactory` is used to create necessary instances of statements and expressions for the AST. The listing

```
1  public interface ASTFactory {
2
3    public static interface Tree {
4      // tree traversal methods; e.g.
5      public Tree getUp();
6      public Tree[] getDown();
7    }
8
9    public static interface Expression extends Tree {
10     // validation
11     public void validate(ErrorHandler errorHandler);
12   }
13
14   public static interface Statement extends Tree {
15     // validation
16     public void validate(ErrorHandler errorHandler);
17     // original source code ref.
18     public int getLine();
19     public String getFileName();
20   }
21
22   public static interface TryCatchBlock {
23     // This pure markup interface
24     // is used to ensure type
25     // compatibility.
26   }
27
28   // the factory methods; e.g.
29   public Statement statementTry(String filename, int line, Scope
         scope, Statement statement, TryCatchBlock catchBlock,
       TryFinallyBlock finallyBlock);
30
31   // the factory utility methods
```

```
32    // to create optional terms; e.g.
33    public TryCatchBlock utilTryCatchBlock(Expression identifier,
          Statement statement);
34  }
```

shows an excerpt of the abstract factory including selected inner interfaces. The statements and expressions mentioned in the `ASTFactory` are defined as static, inner interfaces `Statement` and `Expression` within the definition of the factory. Both interfaces extend a common interface called `Tree`. The use of a factory has the advantage to be able to replace their implementations without touching the grammar. Additionally, mark-up interfaces are used to ensure type compatibility, because during AST construction, sub-parts of the AST are created bottom-up via utility methods. These parts are collected and passed to the corresponding parent rule. For example, the AST of the Euclides code

```
1  try {
2    doSomething();
3  } catch(exception) {
4    repairSomething();
5    print("caught exception " + exception);
6  }
```

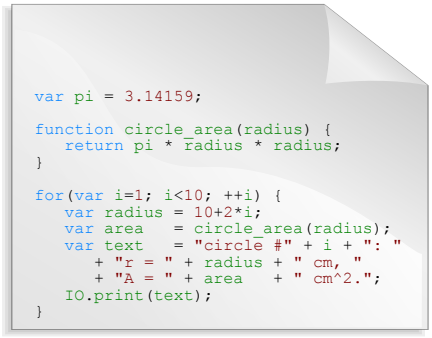
is created via the following factory calls. The optional catch block is parsed by a sub-rule with actions, which call the factory method `utilTryCatchBlock`. This method returns an instance of the mark-up interface `TryCatchBlock`, which can only be passed to a `statementTry` method. This method itself is called in the corresponding rule to match a try statement. In this way, complex grammar rules are split up into several simpler rules while using the abstract factory pattern and maintaining type safety.

The signature of the `statementTry` call reveals some properties that are passed to the factory by all statements: the source code's file name and line together with the current scope. In case of `statementTry`, the statement to try, the optional catch block as well as the optional finally block are also passed to the factory. Note that at least one optional block must be non-null.

The resulting AST is the main data structure for the next stage: semantic analysis. Once all statements and expressions of the input source code are represented in the AST, a tree walker analyzes their semantic relationships, i.e., errors and warnings are generated, for instance, when symbols are used but not defined, or defined but not used.

Having performed all compile-time checks, a translator uses the AST to generate platform-specific files. In other words, this task involves complete and accurate mapping of the AST to constructs of the target platform.

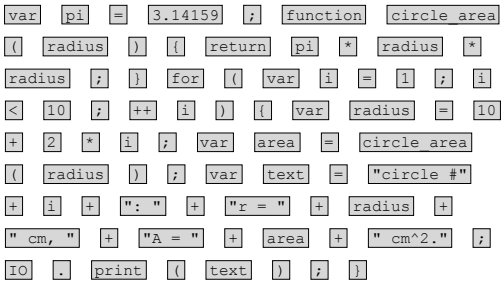
The example shown in Figures 4.3 and 4.4 shows a compilation process of JavaScript. In JavaScript, the top-level rule of an AST is always a simple list of statements – no enclosing class structures, no package declaration, no inclusion instructions, etc. Each statement contains all included substatements and expressions as well as associated comments. During the validation step, this tree structure is extended by reference and occurrence links; e.g., each method call references the method's definition and each variable definition links to all its occurrences. Having assured that all compile-time checks are carried out, symbols are stored in a so called namespace. During validation, this data structure is used to detect name collisions (e.g., redefinition of variables) and undefined references (e.g., usage of undeclared variables).



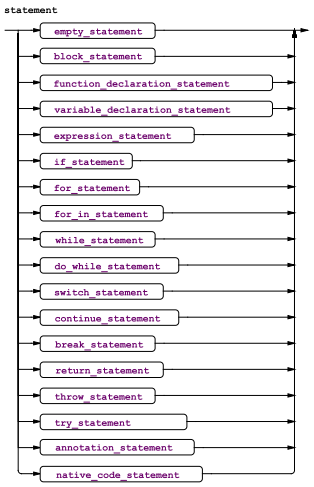
```
var pi = 3.14159;

function circle_area(radius) {
   return pi * radius * radius;
}

for(var i=1; i<10; ++i) {
   var radius = 10+2*i;
   var area   = circle_area(radius);
   var text   = "circle #" + i + ": "
      + "r = " + radius + " cm, "
      + "A = " + area   + " cm^2.";
   IO.print(text);
}
```

1. The first step in a compiler pipeline is performed by a *lexical analyzer*. It converts source code (top left) into a sequence of tokens, i.e., a string of one or more characters that is significant as a group. Tokens are identified based on specific rules of the *lexer*.

2. The stream of tokens (middle right) is processed by the *syntactic analyzer* based on the grammar rules of the language. This example is written in JavaScript. Its entry point into the grammar rules is a `statement`-rule.



```
var pi = 3.14159 ; function circle_area
( radius ) { return pi * radius *
radius ; } for ( var i = 1 ; i
< 10 ; ++ i ) { var radius = 10
+ 2 * i ; var area = circle_area
( radius ) ; var text = "circle #"
+ i + ": " + "r = " + radius +
" cm, " + "A = " + area + " cm^2." ;
IO . print ( text ) ; }
```



3. The result of the parsing step is an *abstract syntax tree* (AST) (see Figure 4.4 right). Afterwards, the *semantic analyzer* constructs the table of symbols (see Figure 4.4 left) and generates all references for resolving names (see Figure 4.4 red pointers).

4. In an optional step, an *optimizer* makes changes to the AST to speed up the final code; in this example, the variable `pi` and the function `circle_area` are only assigned once and can be resolved and inlined in order to reduce the number of look-ups and function calls.

5. In the final step a *code generator* produces object code of the target platform.

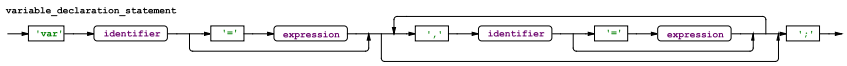


Figure 4.3: A compiler consists of three main components: a *front-end* reads in the source code and constructs a language-independent representation – a so-called abstract syntax tree (AST). The *middleware* performs normalization and optimization steps on the AST. Finally, the *back-end* generates platform-specific object code, i.e., executables, libraries, etc.

Table of symbols:

GLOBAL.IO
[...]

GLOBAL.area
[...]

GLOBAL.circle_area
[...]

GLOBAL.circle_area.msg
[...]

GLOBAL.circle_area.radius
[...]

GLOBAL.i
[...]

GLOBAL.pi
declaration statement:
`var pi = 3.14159;`
declaration file & line:
example.ecs:1
references:
1. example.ecs:1
   `var pi = 3.14159;`
2. example.ecs:4
   `return pi * radius * radius;`

GLOBAL.radius
[...]

GLOBAL.text
[...]

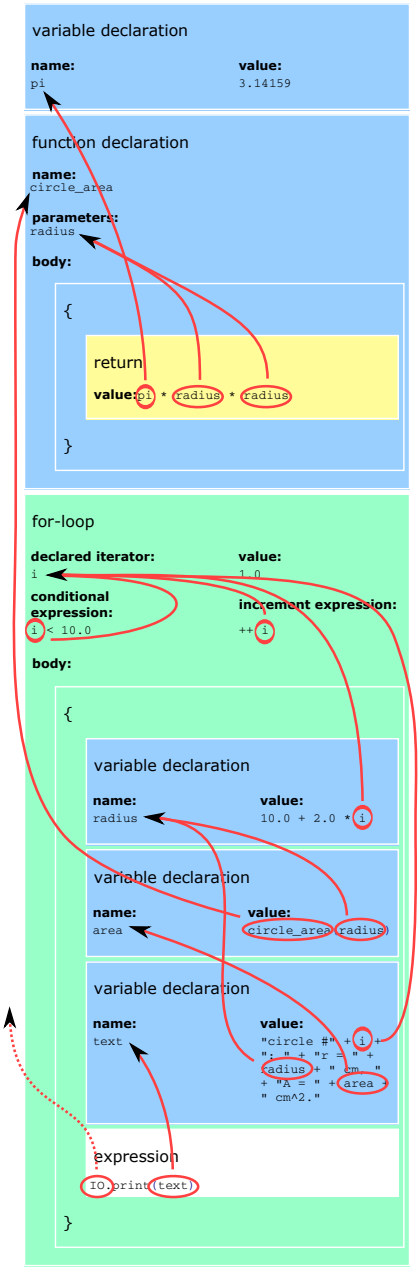


Figure 4.4: A very important data structure within a compiler is the abstract syntax tree (AST), which represents the input source code in a language-independent way. It consists of a tree structure to encode the hierarchical, nested statements (right) enriched by references to resolve symbols (visualized in red). Additional data structures – such as a table of symbols (left) – simplify the work performed by the compiler's middleware.

4.3 Language Elements

In favor of an easy to use language we decided to omit unnecessary language features of the JavaScript specification (ECMAScript 262 [Int15]). For example, the concepts of prototypes, iterators, arrow function definitions, modules or classes are not needed in the context of generative modeling. Regarding data types, the ECMAScript 262 standard lists `undefined`, `null`, `boolean`, `string`, `symbol`, `number`, or `object` (Please note that arrays are treated as regular objects) [Int15]. Euclides has direct support for the following data types as first class citizens (while maintaining compatibility): `undefined`, `boolean`, `number`, `string`, `array`, `object`, or `function`.

While it is our aim to be compliant to the standard, we try to not to add new language constructs and features, which would result in errors when using a standard JavaScript engine. During the development process the compiler's conformance with the JavaScript standard is tested with JavaScript engines of various web browsers.

Due to the nature of the meta-modeling approach we have to differ from the standard in two aspects (but do not create errors with standard JavaScript engines). The first difference is related to the fact that our system is relying on a runtime environment in the target language. In order to enable the execution of platform specific functionality – like text output – it is necessary to include native code within the Euclides syntax. This is done via a special statement called native code statement (see Appendix A.2.18). A native code statement looks like a normal JavaScript multi-line comment with the difference that it is starting with `/*%` instead of `/*` followed by the native code of the target platform and ending with `*/` like in JavaScript. For ECMAScript 262 compliant compilers native code statements still look like JavaScript's multi-line comments.

The second difference affects comments. While short comments start with `//` (like in JavaScript), multi-line comments start with `/**` instead of JavaScript's `/*`. This is a necessary adaptation to be able to distinguish between comments, native code statements and another language feature that is currently unused – annotations (see Appendix A.2.17). Similar to native code statements, for ECMAScript 262 compliant compilers multi-line comments and annotations still look like JavaScript's multi-line comments.

A complete description of all language elements of Euclides can be found in Appendix A.

4.4 Target Platforms

Several targets (target platforms) – each with a different purpose – are available in Euclides (see Figure 4.1):

- The documentation target offers a standard XML representation for long-term archival.
- The modeling power of the GML (see Section 3.4) can be used within Euclides. This target is especially interesting because of the solutions to overcome the major differences in the programming paradigm between Euclides and GML.

- The Java target with 3D output is a sophisticated platform due to the fact that Euclides itself is mainly written in Java.
- A publishing and distribution platform is available with the HTML5 & WebGL target
- A target generating differentiated code is used for numerical optimization, e.g., fitting generative models to real-world data (scanned surfaces).

4.4.1 Documentation Target

The Euclides documentation target (see Figure 4.5) aims to make an important step towards sustainability in procedural modeling by providing a XML representation. The included expert knowledge within an object description (see Section 2.6) is required, for example, in the context of cultural heritage, engineering and manufacturing, or, more generally, by digital library services. For that purpose JavaScript source code is represented as XML structure. Key advantages of the XML format are that it is well-organized, searchable and human readable [Nic02]. Meaningful information needed to perform a source code analysis is also generated during the translation.

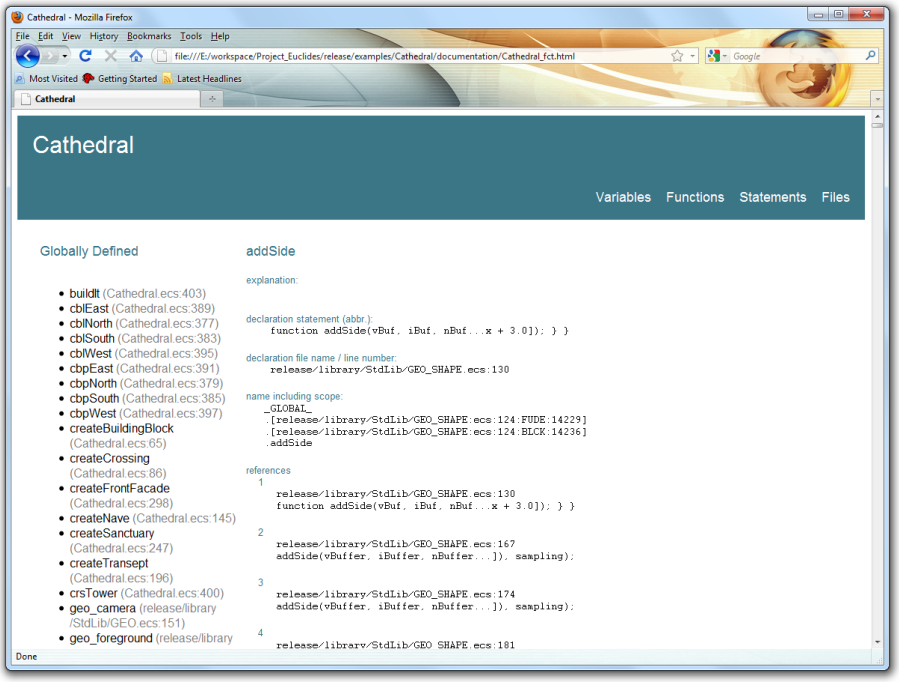


Figure 4.5: The Euclides documentation target, which represents JavaScript as a sustainable, standard-conform XML document can be displayed in an arbitrary web browser.

Four integrated views on the source code are available: Variables, Functions, Statements and Files. All variables (global and local) are listed in the Variables view. For each variable, the following additional information is available:

- **Comments:** Any comments associated with a variable are preserved and included.
- **Location:** The line of code (source, its line number and file name) where the variable is declared.
- **Visibility:** The name together with the scope, in which the variable is available.
- **References:** All references and uses of the variable in the source code including file name, line number and declaration statement.

In the style of the Variables view, the Functions view is a collection of all functions defined in the source code and also offers the additional information mentioned above. The Statements view is a collection of all statements of the source code together with file name and line number. This view allows, for example, identifying duplicate code snippets. Also it gives a nice overview of the complexity of the source code. In the files view, the source code is available as XML document.

4.4.2 GML target

In the articles "Euclides – A JavaScript to PostScript Translator" [SSUF10b] by Martin Strobl, Christoph Schinko, Torsten Ullrich and Dieter W. Fellner and "Scripting Technology for Generative Modeling" [SSUF11b] by Christoph Schinko, Martin Strobl, Torsten Ullrich and Dieter W. Fellner, we describe the translation mechanism to the GML (see Section 3.4).

The GML target significantly reduces the inhibition threshold of using the GML. Even advanced GML users, who already know how to program in PostScript style, can use Euclides to translate algorithms, which are often presented in a imperative, procedural (pseudo-code) style [CSLR01].

Data Types

Variables in Euclides (and JavaScript) have a particular, but dynamic type. Euclides supports the following data types: `undefined`, `boolean`, `number`, `string`, `array`, `object`, or `function`. The GML also offers a dynamical type system. Unfortunately, both type systems are incompatible to each other. Therefore, a direct mapping of Euclides data types to GML data types poses problems. On the one hand, the dynamic types must be inferred at run time. On the other hand, GML's native data types lack distinct features needed by Euclides. GML strings, for example, cannot be accessed character-wise, like Euclides strings. We solved this, and similar problems by implementing Euclides variables as dictionaries [Hav05] in the GML system translation library (GML runtime). Dictionaries are objects that map unique keys to values. We use these dictionaries to hold needed meta data and type information as well as methods, which emulate Euclides behavior. We utilize GML's dictionaries for scoping as well.

The GML runtime used by a translated Euclides program contains the function `sys_init_data`, which defines an anonymous data value in the sense of Euclides data.

```
/sys_init_data {
  dict begin
  /content dict def
  content begin
    /type edef
    /value edef
    /length { value length } def
  end
  content
  end
} def
```

The function `sys_init_data` opens a new variable scope by defining a new, anonymous dictionary and opening it. In this new scope, another newly created dictionary is defined by the name `content`. This content dictionary receives three entries: `type`, `value` and the method `length`. Each entry value is taken from the top of the GML's stack. The newly created dictionary is then pushed back onto the stack and the current scope is destroyed by closing the current dictionary, leaving the anonymous dictionary on the stack. In the GML notation, the content of a Euclides variable is defined by pushing the actual value and a pre-defined constant to identify the type of the variable (such as `Types.number`, `Types.array`, etc.) onto the stack, and calling `sys_init_data`. Consequently, the Euclides statement

```
var foo = 42;
```

translates to

```
/usr_foo 42.0 Types.number sys_init_data def
```

The translator prefixes all Euclides identifiers with `usr_` (in order to ensure that all declarations of identifiers do not collide with predefined GML objects) and uses the following translations:

Undefined: Variables of type `undefined` result from operations that yield an undefined result or by declaring a variable without defining it. The Euclides statement

```
var x;
```

leads to `x` being of type `undefined`. It is translated to

```
/usr_x Nulls.Types.undefined
  Types.undefined sys_init_data def
```

Boolean: In Euclides, Boolean values are denoted by the keywords `true` and `false`. The translation simply maps these values to equivalent numerical values, since the GML does not explicitly support Boolean values. The Euclides statement

```
var x = true;
```

becomes

```
/usr_foo 1 Types.bool sys_init_data def
```

Number: All Euclides numbers (including integers) are represented as 32-bit floating point values. As the GML stores numbers as 32-bit floats internally as well, we simply map them to GML's number representation. A Euclides variable `x` holding a number

```
var x = 3.14159;
```

is translated to

```
/usr_x 3.14159 Types.number sys_init_data def
```

String: Although the GML does support strings, they cannot be accessed character-wise, like in Euclides. We cope with this limitation by defining strings as GML arrays of numbers. Each number is the Unicode of the respective character. As the GML allows to retrieve and to set array elements based on indexes, this approach is a sufficient translation of Euclides strings. The statement

```
var x = "Hello World";
```

becomes

```
/usr_x
  [ 72 101 108 108 111 32 87 111 114 108 100 ]
  Types.string sys_init_data def
```

Array: Euclides arrays allow to hold data of different types – the array's contents may be of mixed types. This behavior can be observed in the GML as well. The Euclides example

```
var x = [true, false, "maybe"];
```

is translated to

```
/usr_x [ 1 Types.bool sys_init_data
         0 Types.bool sys_init_data
         [109 97 121 98 101]
         Types.string sys_init_data ]
         Types.array sys_init_data def
```

Object: In Euclides, an object consists of key-value pairs, e.g.,

```
var x = {
  x : 1.0,
  y : 2.0,
  z : 42
};
```

This structure is mapped to nested GML dictionaries, since the value of a variable's content is a dictionary of its own. This dictionary contains the entries corresponding to entries of the Euclides object, which are also defined as variable contents.

The example above defines a Euclides object of name `x` with key-value pairs `x` to be 1, `y` to be 2, and `z` to be 42:

```
/usr_x dict begin
  /obj dict def obj begin
   /usr_x  1.0 Types.number sys_init_data def
   /usr_y  2.0 Types.number sys_init_data def
   /usr_z 42.0 Types.number sys_init_data def
  end obj
Types.object sys_init_data end def
```

Opening an anonymous dictionary in the GML creates a new scope. In this scope, a dictionary is created and bound to the name `/obj`. It is then opened and its members are defined, just like anonymous variables would be. The object dictionary is then closed, put on the stack, and used to define an anonymous variable. The enclosing anonymous scoping dictionary is then closed and simply discarded.
Euclides objects can hold functions as well. The translator handles Euclides object functions like ordinary functors (referenced functions stored in variables) and assigns their internal name to a key-value pair.

Function: Since Euclides has first-class functions, it is possible to assign functions to variables, which can be passed as parameters to other functions. In this example,

```
function do_nothing() {
  // ... definition of function omitted ...
}
var x = do_nothing;
```

a function `do_nothing` is declared and defined. Afterwards, the function is assigned to a variable `x`. Disregarding the translation of the function `do_nothing`, the assignment is translated to:

```
/usr_do_nothing {
  %% ... definition of function omitted ...
} def

/usr_x
  /usr_do_nothing Types.function
sys_init_data def
```

In Euclides, `x` can be used as a functor, which acts the same ways as`do_nothing`. Because such functors can be reassigned, it is necessary to handle functor calls (`x()`) differently than ordinary function calls (`do_nothing()`). In this situation, Euclides creates a temporary array, which contains the functor parameters and passes this array, as well as the variable referencing the function name, to a system function `sys_execute_var`. This system function resolves the functor and determines the referenced function, unwraps the array and performs the function call.

Functions

In the GML, functions are defined using closures, such as `/my_add { add } def`. If the function `my_add` is executed, the closure `{ add }` is pushed onto the stack, its brackets are removed, and the content is executed.

To execute a GML function, its parameters need to be pushed onto the stack prior to the function call: `1.0 2.0 my_add`. The resulting number `3.0` will remain on the stack. GML functions may, very well, produce more than one result (left

on the stack) at each function call. This allows to define functions with more than one result value. In contrast to the GML, Euclides functions, by convention, only return one value. The number and names of function parameters are known at compile time. Only functors may change at run time and cannot be checked ahead of time.

Translated functions and parameters are named just like their Euclides counterparts (except for their `usr_` prefix to avoid namespace collisions).

Scopes: As Euclides uses a scoping mechanism that differs from the GML mechanism, it has to be emulated. This is a rather complex task, taking into account the following properties of Euclides scopes:

- Euclides functions can call other functions or themselves.
- Called functions can declare the same identifiers as the calling functions.
- Within functions other functions can be defined.
- Blocks can be nested inside functions, redefining symbols or declaring symbols of the same name.

The translator uses GML's dictionary mechanism to emulate Euclides scopes. A dictionary on the dictionary stack can be opened and it will take all subsequent assignments to GML identifiers (variables). Since only the opened dictionary is affected, this behavior is identical to opening and closing scopes in different scoped programming languages, such as C or Java.

Because of that property, an assignment `/x 42 def` can be put into an isolated scope by creating a dictionary (`dict`), opening it (`begin`), performing the assignment, and closing the dictionary (`end`). The following example shows how such GML scopes can also be nested:

```
1 dict begin
2   /x 3.141 def            %% x is 3.141
3   dict begin              %%
4     /x 4 def              %% x is 4.0
5   end                     %% x is 3.141
6 end                       %% x is unknown
```

As noted before, Euclides supports redefinition of identifiers that were declared in a scope below the current one. The GML exhibits just the same behavior when reading out the values of variables (keys) from dictionaries of the dictionary stack. Consequently, the following example works as expected:

```
1 dict begin
2   /x 42 def
3   dict begin
4     /y x 1 add def        %% y is now 43
5   end
6 end
```

However, assignments to variables have to be handled differently in the GML. The GML does not distinguish between declaration and definition. Any declaration must be a definition and vice versa.

To solve this problem, the translator uses a system function called `sys_def`, which is automatically included in translated Euclides code. This function uses the GML's `where` operator, which, applied on the dictionary stack, finds the

uppermost dictionary defining a searched name. The operator returns the reference to the dictionary, in which the name was found.

Control Flow for Functions: The GML, as well as all PostScript dialects, lack a dedicated jump operation in control flow. Imperative functions often require the execution context to jump to a different point in the program at any time – and to return from there as well.

The GML's exception mechanism can be used to emulate this behavior. A GML exception is propagated down the GML's internal execution stack until a `catch` instruction is encountered. On its way it overrides any other control structure it encounters. We use the GML's exception mechanism to jump outside a function as illustrated in the following empty function skeleton:

```
1  /usr_foo {
2    dict begin
3    /return_issued 0 def
4    { dict begin
5      %% ... function body omitted ...
6      end }
7    { /return_issued 1 def }
8    catch
9
10   return_issued not
11   { Nulls.Types.undefined
12     Types.undefined sys_init_data } if
13   end
14   sys_exception_return_handler
15 } def
```

In this empty skeleton, the function opens a new anonymous scope. Inside this scope `dict begin ... end` the local identifier `/return_issued` is set to `0`. Afterwards a GML try-catch statement { *try_block* } { *catch_block* } `catch` contains the Euclides function implementation. In this translation, the catch block redefines `/return_issued` to `1` to indicate that a Euclides `return` statement has been executed in the function body. Euclides functions without any `return` statement automatically return `null`, respectively `Nulls.Types.undefined Types.undefined sys_init_data`. A corresponding Eculides return statement, e.g.,

```
1  return 42;
```

is translated to

```
1  42.0 Types.number sys_init_data end throw
```

In this example, the number `42.0` is pushed onto the stack. The actual function body's scope is closed with `end`, and the `throw` operator is applied. The distinction of whether the end of the function body was reached by normal program flow or via a return statement determines, if a return value needs to be constructed (`null`) and pushed onto the stack.

Parameters to functions are simply pushed onto the stack. The function body retrieves the expected number of parameters and assigns them to dictionary entries of the outer scope defined in the function translation. A complete example of a translated Euclides function shows the interplay of all mechanisms. The simple Euclides function

```
function foo(n) {
  return n;
}
```

is translated to

```
/usr_foo {
  dict begin
  /usr_n edef
  /return_issued 0 def
  { dict begin
    usr_n
    end
    throw
    end }
  { /return_issued 1 def }
  catch

  return_issued not
  { Nulls.Types.undefined
    Types.undefined sys_init_data } if
  end
  sys_exception_return_handler
} def
```

A function call, for example `foo(3)`, yields the translation `3.0 Types.number sys_init_data usr_foo`. If we assign the function `foo` to a variable `foo_functor`, the calling convention in the GML changes significantly.

```
/usr_foo_functor
  /usr_foo Types.function sys_init_data def
```

is called via

```
[ 3.0 Types.number sys_init_data ]
usr_foo_functor sys_execute_var
```

and represents the Euclides call

```
foo_functor(3.0);
```

Exceptions

Euclides offers support for throwing exceptions as shown in the following example:

```
throw "Error: unable to read file.";
```

Its syntax is similar to a return statement. To implement such behavior, we also use the GML's exception handling mechanism. The Euclides translator adds a call to the predefined system function `sys_exception_return_handler` at the end of each translated function (see example above).

Throwing an exception in Euclides translates into a global GML variable `exception_thrown` being set to 1, closing the current dictionary and calling the GML's `throw`. The `sys_exception_return_handler` checks if an actual exception is being thrown, and if so, calls `throw` again. A catch block inside a Euclides program sets `exception_thrown` to 0.

Operators

The evaluation of expressions demands variables to be accessed. While the GML provides operators that operate on their own set of types, they obviously cannot be used to access the translated Euclides variables. For this reason, the Euclides translator automatically includes a set of predefined GML functions that substitute operators defined in JavaScript.

Value Access: By performing the opposite operation to `sys_init_data`, the operation `sys_get_value` will retrieve the data saved in a Euclides variable, respectively its GML dictionary. For example, to retrieve `v.value` the function `sys_get_value` is applied to `v`.

```
/sys_get_value { begin value end } def
```

Element Access: The system function `sys_get` implements string, array and object access. Applied to a string or an array (`Arr` and index `k`), it will return the element `Arr[k]`. If its parameters are an object `Obj` and an attribute `name`, the function `sys_get` executes `Obj.name`. This results in a value, which is pushed onto the stack or in a function, which is called. In conformance with Euclides, it returns Euclides `undefined` for any requested elements that do not exist.

```
/sys_get {
  dict begin
  /idx exch def /var exch def

  var.type Types.string eq {
    %% ... handling strings ...
  } if

  var.type Types.array  eq {
    %% ... handling arrays  ...
  } if

  var.type Types.object  eq {
    var sys_get_value idx known 0 eq {
      %% return null, if element
       %% does not exist
       Nulls.Types.undefined
       Types.undefined sys_init_data
    } if
    var sys_get_value idx known 0 ne {
      %% access element
      var sys_get_value idx get
    } if
  } if
  end
} def
```

In analogy to `sys_get`, `sys_put` inserts data into strings and arrays, or defines members of objects. If `sys_put` encounters an index k that is out of an array's range, the array is resized and filled with Euclides `undefined`s.

Functors: The already mentioned routine `sys_execute_var` inspects a given variable. If it is a function, it retrieves the array supplied to hold all parameters and executes the function. However, the dynamic binding of functions to

variables requires to consider two situations at run time: The functor receives the correct amount of parameters for its function, or the number of parameters does not correspond to the referenced function. In the latter case, the function is not called and `null` is returned instead.

At compile time, a function is defined to expect a concrete number of parameters. This information is kept to perform parameter checks at run time. In this way, the correct number of parameters for all functors can be determined any time.

```
1  variable.type Types.function eq
2  {
3    %% count the stacksize prior to parameter unwinding
4    count /stacksize_old edef
5    100
6      { Nulls.Types.undefined Types.undefined sys_init_data }
7    repeat                  %% create excessive "dummy" parameters
8    params {} forall        %% puts all parameters on the stack
9    variable.value cvx exec %% converts to executable literal and
         executes
10
11   count /stacksize_new edef
12   %% Fix stack size, saving result and discarding any excessive
         parameters
13   %% caller may have specified:
14   /res edef
15   stacksize_new stacksize_old sub
16   1 sub
17   { pop } repeat
18   res
19 }
20 if
```

To calculate the size of the stack prior to a function call, we push the parameters specified at runtime onto the stack, execute the function, and count the stack size again. We can then calculate the number of parameters the function expected. Afterwards, we save the function's return value, pop any unconsumed parameters off the stack, and push the return value onto the stack again.

The opposite case of insufficient specified parameters to a function is countered by pushing an arbitrary number of JavaScript `undefined`s onto the stack prior to the actual function's parameters.

JavaScript built-in Operators: The translation of relational, arithmetical or bit-shift operators defined by Euclides, is demonstrated by using the equal operator `==`. It is – like all such operators – mapped to a corresponding routine `sys_eq`. Depending on the operands' types, it delegates the comparison to subroutines such as `bool_eq`, `string_eq` or `array_eq` that perform the actual comparison. If the types and the values match, `sys_eq` directly returns the Euclides value `true`. If the types do not match, the variable is converted to the type of the respective operand, as specified by Euclides, and then compared.

Control Flow

Since the Euclides if-then-else statement corresponds one-to-one to the same GML statement, the conditional expression can be translated directly. Using the expression mapping introduced in the previous Section (e.g., `sys_eq` implements the equality operator), the Euclides statement

```
if (a == b) {
  c = a;
} else {
  c = b;
}
```

is translated to

```
%% if (a==b)
usr_a usr_b sys_eq sys_get_value
{ %% then:
  dict begin {
    dict begin
      /usr_c usr_a sys_def
    end
  } exec end
}
{ %% else:
  dict begin {
    dict begin
      /usr_c usr_b sys_def
    end
  } exec end
} ifelse
```

The `exec` statements (and their closures) stem from the fact that both sub-statements, the then-part and the else-part, are statement blocks { ... }. These blocks are executed within their own, new scopes.

Loops: The GML supports different types of looping control structures, which are similar to Euclides loops (e.g., both languages have a for loop). However, the GML counterparts have different semantics. For example, the GML's for-loop has a fixed, finite number of iterations, which is known before execution of the loop body, whereas Euclides loops evaluate the stop condition during execution, which may result in endless loops. The Euclides translator uses the GML `loop` mechanism, which is an infinite loop that can be exited using the `exit` operator.

An important problem is that control structures such as for, while and do-while loops are not only controlled by the loop's stop condition, but also by Euclides statements such as `continue` and `break` within the loop body (besides `return` and `throw` as mentioned before). The statement `break` immediately stops execution of the loop and leaves it, whereas `continue` terminates the execution of the current loop iteration and continues with the next iteration of the loop. Therefore, we translate an empty while loop

```
while (false) {
  // ... loop body omitted ...
}
```

to

```
{  /continue_called 0 def
  {  0 Types.bool sys_init_data
    sys_get_value not { exit } if
    {  dict begin
      %% ... loop body omitted ...
      end
    } exec
  } loop
  continue_called not { exit } if
} loop
```

The GML's `exit` keyword terminates the current loop. This behavior is leveraged by the Euclides translator to implement `break` and `continue`. The translation uses two nested loops that will run indefinitely.

Prior to the start of the inner loop, `/continue_called` is set to `0`. Then, the loop condition is tested. If the condition evaluates to `false`, the inner loop is exited using the GML's `exit`. Otherwise a new scope is created and the loop statement executed within that scope.

During loop iterations, there are three scenarios under which a loop can terminate:

1. If the loop condition is met: When the condition evaluates to `false`, the inner loop is exited. Since `continue_called` is not set to `true`, the outer loop will terminate as well.

2. If the loop body encounters JavaScript `break` (resp. GML `exit`): Again, the inner loop is left. `continue_called` will not be set to `true`, hence the outer loop will also terminate.

3. If the function returns: The GML's exception throwing mechanism will unwind the stack until the catch-handler at the end of the function is encountered.

If the loop body encounters a Euclides `continue` statement, `continue_called` will be set to `true` and the GML `exit` command will immediately stop the inner loop. Since `continue_called` is set, execution does not leave the outer loop, however. As a consequence, `continue_called` becomes `0` again, and execution re-enters the inner infinite loop.

The do-while statement is translated like the while statement. The only semantic differences in Euclides are that execution will enter the loop regardless of the loop condition and that the loop condition is tested after loop body execution. Euclides translates an empty do-while statement

```
do {
  // ... loop body omitted ...
} while (false)
```

as follows:

```
{  /continue_called 0 def
  { {  dict begin
       %% ... loop body omitted ...
       end
    } exec
    0 Types.bool sys_init_data
      sys_get_value not { exit } if
  } loop
  continue_called not { exit } if
  0 Types.bool sys_init_data
  pop
} loop
```

Due to a semantic difference of Euclides `continue` in do-while loops, this statement needs to be handled differently. If `continue` is encountered, the loop condition must still execute before the loop body is re-entered, because side effects inside the loop condition may occur (such as incrementing a counter). Euclides handles this problem by executing the condition expression a second time in the

outer loop. Since expressions always return values, any value resulting from the loop-expression has to be popped off the stack.

Although the GML has a `for` operator, it is semantically incompatible with Euclides' one. Its increment is a constant number, and so is the limit. In Euclides, both, increment and limit, must be evaluated at each loop body execution. Therefore, we translate `for` just like the previous constructs by two nested loops with the increment condition repeated in outer loop (due to `continue` semantics). Euclides translates the statement

```
for (var i = 0; i < 1; i++) {
  // ... loop body omitted ...
}
```

to GML via

```
dict begin
%% initialization (i=0)
/usr_i 0.0 Types.number sys_init_data def
{  /continue_called 0 def
  {  %% condition (i<1)
    usr_i 1.0 Types.number
    sys_init_data sys_lt
    sys_get_value not { exit } if
    {  dict begin
      %% ... loop body omitted ...
      end
    } exec
    %% increment (i++)
    usr_i
      usr_i 1 Types.number
      sys_init_data sys_add
    /usr_i sys_edef
    pop
  } loop
  continue_called not { exit } if
  %% increment again (i++)
  usr_i
    usr_i 1 Types.number
    sys_init_data sys_add
  /usr_i sys_edef
  pop
} loop
end
```

In Euclides, the following for-in statement

```
for (var x in array) {
  // ... statement body omitted ...
}
```

is semantically equivalent to:

```
for (var i= 0; i < array.length; i++) {
  var x = array[i];
  // ... rest of statement body omitted ...
}
```

This construction loops over the elements of an array and provides the loop body with a variable holding the current element. Although the GML does provide an operator `forall`, we decided to implement the `for ... in` operator analogous to its `for` variant, because this translation is more safely constructible: `forall` needs to be in the outer loop, which breaks the mechanisms laid out before.

Selection Control Statement: The translation of the Euclides `switch` statement poses several difficulties:

- If a case condition is met, execution can "fall through" till the next `break` is encountered.
- If a `break` is encountered, the currently executed `switch` statement must be terminated.
- Of course, `switch` statements may be nested.

To develop a semantically consistent solution, we did not want to alter the translation of Euclides `break` expression inside switch statements (compared to loops). We solve the problem of breaking outside the `switch` statement by implementing it as a loop that is run exactly once. In the GML it reads like `1 {` *loop_instructions* `} repeat`. This way our translation of `break` shows semantically correct behavior, it terminates the loop. Consider the following Euclides program:

```
1  var x = 0;
2  var y = 0;
3
4  function bar() {
5    return 3;
6  }
7
8  function foo(i) {
9    switch(i) {
10     case 0:
11     case 1:
12     case 2: x = 1;
13     case 4: x = 3;
14     case bar(): x = 2;
15       break;
16     default: y = 5;
17   }
18 }
```

The function `foo` is translated to:

```
1  /usr_foo
2  { dict begin
3    /usr_i edef
4    /return_issued 0 def
5    { dict begin
6      /switch_cnd_met1 0 def
7      1 { usr_i 0.0
8        Types.number sys_init_data
9        sys_eq sys_getvalue
10       switch_cnd_met1 1 eq or {
11         /switch_cnd_met1 1 def
12       } if
13
14       usr_i
15       1.0 Types.number sys_init_data
16       sys_eq sys_getvalue
17       switch_cnd_met1 1 eq or {
18         /switch_cnd_met1 1 def
19       } if
20
21       usr_i
```

```
22        2.0 Types.number sys_init_data
23        sys_eq sys_getvalue
24        switch_cnd_met1 1 eq or {
25          /switch_cnd_met1 1 def
26          %% x = 1;
27          /usr_x 1.0 Types.number
28            sys_init_data sys_def
29        } if
30
31        usr_i
32        4.0 Types.number sys_init_data
33        sys_eq sys_getvalue
34        switch_cnd_met1 1 eq or {
35          /switch_cnd_met1 1 def
36          %% x = 3;
37          /usr_x 3.0 Types.number
38          sys_init_data sys_def
39        } if
40
41        usr_i usr_bar
42        sys_eq sys_getvalue
43        switch_cnd_met1 1 eq or {
44          /switch_cnd_met1 1 def
45          %% x = 2;
46          /usr_x 2.0 Types.number
47          sys_init_data sys_def
48          exit
49        } if
50        %% y = 5;
51        /usr_y 5.0 Types.number
52        sys_init_data sys_def
53      } repeat
54      currentdict /switch_cnd_met1 undef end
55    }
56    { /return_issued 1 def } catch
57
58    return_issued not {
59      Nulls.Types.undefined
60      Types.undefined sys_init_data
61    } if
62    end
63    sys_exception_return_handler
64  } def
```

In this example we introduce an internal variable `/switch_cnd_metX` for traversing the case statements. When a case statement condition is met, `/switch_cnd_metX` is set to `true`, directing execution into every encountered case statement.

The Euclides translator takes into account that switch statements may be nested. As it traverses the AST, it keeps book of all internal variables to ensure a unique name (`switch_cnd_met1`, `switch_cnd_met2`, ..., `switch_cnd_metN`).

The example translation shows that for `foo(3)` the cases 0, 1, 2, 4 and 3 (= `bar()`) will only execute case 3, where the `1 { } repeat` statement will be broken out of with the GML `exit` operator. The default block will be executed in any case if execution is still inside the `repeat` statement, no further state is checked for `default`.

4.4.3 Java target

In the article "Scripting Technology for Generative Modeling" [SSUF11b] by Christoph Schinko, Martin Strobl, Torsten Ullrich and Dieter W. Fellner, we introduce a translation mechanism to Java.

Although Java and JavaScript have some similarities, the concepts of both languages show major differences. Java is a statically typed, class-based, general-purpose programming language designed to have a minimum of implementation dependencies to be able to run on many different platforms.

An important reason why we have chosen Java to be a target language is because all front-end and framework components themselves are written in Java making it easier to be embedded in an integrated development environment.

Data Types

Because of conceptual differences in the typing system, it is impractical to project JavaScript data types onto built-in Java data types. For example, JavaScript makes no difference between integer numbers or floating-point numbers. There is just one data type called `number` that may hold any type of number. Additional differences can be found when comparing the remaining data types. As a consequence, each JavaScript data type is re-built in Java to match its functionality resulting in a total of seven data types (`array`, `boolean`, `function`, `number`, `object`, `string`, `undefined`). These data types are wrapped in a class called `Var`, which provides the properties:

- `getType()`: The method `getType()` returns an enum `VarType` with the type of the variable.
- `length(long ii)`: The method `length(long ii)` returns the length of the variable as a variable.

The access functions are:

- `accessArray(long ii, Var index)`: The method `accessArray(long ii, Var index)` returns the element of an array with the position: `index`.
- `accessObject(long ii, String attribute)`: This method returns the value of an object with the key: `attribute`. Note that it is possible to obtain the length of an array accessing its `length` key.
- `execute(long ii, Var THIS, Var[] parameters)`: This method executes a method passing the *this* reference (the current execution context) as well as possible parameters in the form of an array.

Conversion methods are applicable to all JavaScript variables through the following methods:

- `toArray()`
- `toBoolean()`
- `toFunction()`
- `toNumber()`

- `toObject()`
- `toString()`
- `toUndefined()`

These conversions are performed implicitly, but not all conversions yield a meaningful result, e.g., a conversion to a function results in an empty function. Many of the above methods expect a parameter called `ii`. It always refers to a table entry, which references the corresponding line of JavaScript source code; e.g., each data type can be accessed like an array. In case of an array, the access is "as supposed", in case of a String it is character-wise, in all other cases an implicit conversion creates a new, empty array. As the runtime environment produces warnings if implicit conversions take place, the implementation of an array access includes the statement `Log.variableTypeChangeImplicit(ii);`. In the messages table entry, generated by the compiler, `#ii` references information needed for a reasonable warning; e.g., during the execution of

```
1 var number = 42;
2 number = "Hello World";
```

the runtime environment produces the warning

```
assignment provoked a warning.
  type    : variable type change by assignment
  file    : C//Users/ullrich/warning.ecs
  line    : 2
  details : number = "Hello World";
```

indicating the type change caused by the assignment of *Hello World* in line 2.

The implementation of the JavaScript data types in the Java runtime environment eventually uses mappings to Java data types (in the `Var` classes of the appropriate data types):

- **Boolean**: The `boolean` data type is mapped to the corresponding Java data type `boolean`.
- **Number**: A JavaScript `number` is mapped to `double`.
- **String**: `String` in JavaScript can be mapped to `String` in Java.
- **Array**: A JavaScript `array` is realized using a collection: `ArrayList<Var>`.
- **Object**: An `object` in JavaScript is mapped to a map of key/value pairs: `HashMap<String,Var>`.
- **Function**: The corresponding object to a JavaScript functor is a function pointer implementation in Java via abstract objects.

A consequence of these data type implementations is the necessity to use a runtime environment in the translated Java code. Whenever a variable is created or a value is assigned, a method-call is performed – thus significantly increasing the execution time of the code. However, for creating variables, a factory pattern is applied with the inherent advantage of exchangeability.

Concerning language constructs, a wide range can be translated easily, since they are available in Java and have the same semantic meaning in both languages. Sometimes, there is the need to utilize temporary variables, which implicate a possible naming conflict with variable names used in the original JavaScript source code. This problem is tackled by prefixing all original JavaScript names and additionally creating unique names for temporary variables.

Functions

In Java, invokable routines are called methods and they are similar to, but not quite like functions in JavaScript. The runtime environment provides a class for JavaScript functions to mimic their behavior. An important property of functions in JavaScript is that they can be `undefined`. Therefore, when instantiating an empty function in Java, a *dummy* with the correct behavior is returned. Executing a function in the Java runtime environment is done by calling the `execute` method in the function class. In addition to function parameters, an environment reference is passed to the function in order to enable correct interaction with the immediate environment. Functions extend an abstract class called `Fct` defining all necessary methods:

- `getID()`
- `getName()`
- `getTranslatedName()`
- `getAnnotations()`
- `getParam()`
- `getParams()`
- `execute(long ii, Var THIS, Var[] parameters)`
- `execute(long ii, Var THIS, Var usr_vecArray)`

They reside in a *public*, *final* class called `Function`. Consequently, the function

```
1 function add(a,b) {
2   return a + b;
3 }
```

gets translated to

```
1  @Override
2  public Var execute(long ii, Var THIS, Var usr_a, Var usr_b) {
3    try {
4      {
5        if (Main.AVOID_UNREACHABLE_CODE_ERROR)
6          return Op.ADD(0, usr_a, usr_b);
7      }
8    } catch (EuclidesRuntimeException exp) {
9      throw exp;
10   } catch (RuntimeException exp) {
11     Log.uncaughtException(ii);
12     System.err.println(exp);
13     System.exit(0);
14   }
15   return Factory.initUndefined();
16 }
```

The body of the function is embedded in a try-catch block in order to throw runtime exceptions or halt execution in case of an unhandled exception. The value `undefined` is returned in case of a runtime exception. Note that the static constant `Main.AVOID_UNREACHABLE_CODE_ERROR` is always true and only needed to avoid – as it says – "unreachable code errors" thrown by Java compilers, for example, if a return statement is followed by further statements.

Translated functions and parameters are named just like their JavaScript counterparts (except for the `usr_` prefix).

Operators

Since JavaScript data types are not directly mapped to native Java data types, all operators need to be recreated in the Java runtime environment as well. A total of 35 operators grouped in unary, binary and tertiary operators are available. Since each operator is applied via a method call, they can be easily exchanged. Operators are collected as methods in a *public*, *final* class called `Op`. As an example, the following operation

```
var c = 19.0 + 23.0;
```

results in

```
Variable.usr_c.assign(1, Op.ADD(0, Factory.initNumber(19.0),
    Factory.initNumber(23.0)));
```

The result of the call to `Op.ADD` with the two numbers as parameters is stored in a new variable, which is returned and then used as a parameter for the assignment operation.

Control Flow

Control flow statements are widely identical in both languages. One of the differences, however, is the switch-statement. For a switch statement in Java only primitive data types are allowed, whereas JavaScript allows all types to be used, attributable to dynamic typing. In order to obtain a correct translation, the switch statement needs to be rewritten, which is done directly in the translated code. The first step is to analyze the statement from back to front comparing each case with the switching expression. Then the result is stored in a temporary variable and the switch-statement is rebuilt in reverse order using the temporary variable as switching expression. As a result

```
switch (favoritelanguage) {
  case "Java":
    io_stdout_write("Good choice!");
    break;
  case "C":
    io_stdout_write("Bad choice");
    break;
  default:
    io_stdout_write("I have no idea");
}
```

is translated to

```
int sys_42 = 0;
if (Op.EQ(9, Variable.usr_favoritelanguage, Factory.initString("
    C")).toBoolean())
```

```
3     sys_42 = 1;
4   if (Op.EQ(10, Variable.usr_favoritelanguage, Factory.initString(
        "Java")).toBoolean())
5     sys_42 = 2;
6   switch (sys_42) {
7     case 2:
8       Function.usr_io_stdout_write.execute(11, THIS, Factory.
            initString("Good choice!"));
9       if (Main.AVOID_UNREACHABLE_CODE_ERROR)
10        break;
11    case 1:
12      Function.usr_io_stdout_write.execute(12, THIS, Factory.
            initString("Bad choice"));
13      if (Main.AVOID_UNREACHABLE_CODE_ERROR)
14        break;
15    default:
16      Function.usr_io_stdout_write.execute(13, THIS, Factory.
            initString("I have no idea"));
17  }
```

The corresponding translation in Java creates the temporary variable `sys_42` for comparisons and a switch statement in reverse order to rebuild the behavior of the JavaScript counterpart.

Once all target files containing source code are generated, they are compiled using the Java compiler included in the Java Platform, Standard Edition (Java SE). The resulting class files are automatically packed into a single Java ARchive (JAR) file for easy execution. As a last step, the JAR file is digitally signed to be ready-to-use for Java Web Start. The signature information becomes part of the embedded manifest file.

4.4.4 HTML5 & WebGL target

The following work has been implemented in the context of the bachelor thesis *Generative Modelling and HTML5* by Franz Papst at Graz University of Technology under the supervision of Torsten Ullrich and Christoph Schinko.

The HTML5 & WebGL target offers the ability to translate Euclides code in a way that it is embedded in HTML5 output for execution in a web browser. However, since HTML5 is a markup language it can only provide a framework for interaction. For scripting purposes, it specifies APIs that can be used with JavaScript. At first glance it may seem odd to write a Euclides to JavaScript translator, since Euclides code can be directly evaluated by a JavaScript interpreter. While this is the case when we just look at the language constructs themselves, even simple examples need to interact with the environment they are executed in, e.g., basically every generated output file needs basic functionality for input & output. As a result, a runtime environment must provide this functionality. More complex code with graphical output needs a lot more functionality provided by the runtime environment (e.g., the HTML5 canvas element, functionality to handle mouse interaction, or fragment and vertex shaders, ...) before the generated output is able to render anything using WebGL.

Having established the fact that Euclides language constructs can be directly mapped onto JavaScript constructs, the translation process is simplified. Without the need to put much effort into translating language constructs, the translated code must be embedded into a suitable runtime environment.
An important step during translation is to avoid namespace collisions between

the runtime environment and the translated code. Similar to other translation targets, this is solved by adding the unique prefix `usr_` to all the function and variable names in the translated code. A function `f(x)` in Euclides code

```
function f(x) {
  return x * 42;
}
```

gets translated to

```
function usr_f (usr_x) {
  {
    return usr_x * 42.0;
  }
}
```

Apart from the prefixing of variable and function names, or the introduction of additional blocks, the translation process offers no surprises. Note that also the code of provided libraries (like GUI or MATH, see Section 4.5) is treated as user-generated code and gets translated to JavaScript.

Runtime environment

The runtime environment consists of a HTML5 template that is used to create an output file incorporating the translated user-generated code as well as all included libraries. A `main` function is the starting point of the output file. It gets called automatically using the `onload` functionality of the HTML5 <body> element. The output is then populated with the translated statements of the user-generated code as well as the library code.

Since Euclides' main purpose lies in generative modeling, providing functionality to display 3D graphics is an important aspect of the runtime environment.

3D Graphics The visualization of 3D content is done using WebGL and the HTML5 <canvas> element. The <canvas> element was introduced in the HTML5 standard to display (interactive) multimedia content without the need to use external plug-ins like Adobe Flash or Microsoft Silverlight. Apart from its extensive 2D abilities, the <canvas> element can also be used to display 3D content using WebGL. WebGL 1.0 is based on OpenGL ES 2.0 and provides an API for 3D graphics. A major difference to OpenGL versions prior to 3.0 is the mandatory usage of buffers – it is no longer possible to use the `glBegin()` and `glEnd()` functions for drawing. Also the usage of vertex and fragment shaders is mandatory in WebGL. The runtime environment reflects the need to use shaders with the following vertex shader:

```
attribute vec4 aVertexColor;
attribute vec3 aVertexPosition;
attribute vec3 aVertexNormal;
attribute vec2 aTextureCoord;

uniform bool uHasTexture;
uniform bool uHasColor;

uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;
uniform mat4 uNMatrix;

varying vec2 vTextureCoord;
```

```
14  varying vec4 vColor;
15
16  void main(void) {
17    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition,
          1.0);
18    if (uHasTexture) {
19      vTextureCoord = aTextureCoord;
20    } else if (uHasColor) {
21      vColor = aVertexColor;
22    } else {
23      vColor = vec4(1.0, 1.0, 1.0, 1.0);
24    }
25  }
```

Apart from all necessary matrices and vertex attributes, the vertex shader calculates texture coordinates, or vertex colors, for the fragment shader stage:

```
1  precision mediump float;
2
3  varying vec2 vTextureCoord;
4  varying vec4 vColor;
5
6  uniform sampler2D uSampler;
7  uniform bool uHasTexture;
8
9  void main(void) {
10   if (uHasTexture) {
11     gl_FragColor = texture2D(uSampler, vec2(vTextureCoord.s,
           vTextureCoord.t));
12   } else {
13     gl_FragColor = vColor;
14   }
15 }
```

The code of the shaders is embedded in the output HTML5 file and gets compiled and linked during the initialization of the 3D environment in the browser. Note that these shaders are rather simple, but can be easily extended as new features of the target become available.

However shapes are created (by using high-level libraries, or by directly creating low-level buffers), the runtime environment relies on buffers to display 3D content. The function `drawScene()` iterates over all active shapes and binds the buffers prior to issuing the WebGL draw command. The previously described vertex and fragment shaders replace a fixed-function pipeline and deal with texturing (or coloring) and lighting of the content.

On a side note, the JavaScript built-in function `setInterval()` is used to call `drawScene()` every 15 milliseconds. This feature of JavaScript replaces the need for an infinite loop.

4.4.5 Differential Java target

In the article "Scripting Technology for Generative Modeling" [SSUF11b] by Christoph Schinko, Martin Strobl, Torsten Ullrich and Dieter W. Fellner, we introduce a translation mechanism to Differential Java.

Besides the previously described targets, Euclides offers a Differential Java target used to compute derivatives of functions. This is a necessary task in many applications of scientific computing, e.g., validating reconstruction and fitting results of laser scanned surfaces [SUSF11], [UF11]: In combination with

variance analysis techniques (see Section 5.2), generative descriptions can be used to validate reconstructions. Detailed mesh comparisons can reveal smallest changes and damages. These analysis and documentation tasks (see Section 4.4.1) are not only needed in the context of cultural heritage but also in engineering and manufacturing. The Euclides framework is used to implement generative models, whose accuracy and systematics describe the semantic properties of an object; whereas the actual object is a real-world data set (laser scan or photogrammetric reconstruction) without any additional semantic information.

This analysis task needs derivatives of the distance-based objective function as well as the embedded generative descriptions. According to Rolf Hammer et al. [HHKR97], there are three different methods to obtain values of derivatives:

- **Numerical differentiation** This method uses difference approximations to compute approximations of the derivative values.
- **Symbolic differentiation** Explicit formulas for the derivative functions are computed by applying differentiation rules.
- **Automatic differentiation** This method also uses the well-known differentiation rules, but it propagates numerical values for the derivatives.

The Differential Java target uses automatic differentiation to obtain derivatives. This is done by replacing variables and operators in the runtime environment, which is an easy task, since variables and operators are created using the factory pattern. The following listing shows the differences between standard and differential multiplication operator. As expected, the standard operator returns a variable initialized with the result of the multiplication operation.

```
1  /**
2   * Binary operator multiply.
3   *
4   * @param ii Information index.
5   * @param v1 The first operand.
6   * @param v2 The second operand.
7   * @return The result.
8   */
9  public static Var MUL(int ii, Var v1, Var v2) {
10   if (!v1.getType().equals(Type.NUMBER)
11    || !v2.getType().equals(Type.NUMBER)) {
12     Log.deviantOperatorCallNoNumber(ii);
13   }
14
15   return Factory.initNumber(v1.toNumber() * v2.toNumber());
16 }
```

The differential operator calculates the derivatives of the operands and stores them in an array. A resulting array is constructed out of the calculated derivatives and returned as a variable.

```
1  /**
2   * Binary operator multiply.
3   *
4   * @param ii Information index.
5   * @param v1 The first operand.
6   * @param v2 The second operand.
```

```
 7   * @return The result.
 8   */
 9  public static Var MUL(int ii, Var v1, Var v2) {
10    if (!v1.getType().equals(Type.NUMBER)
11     || !v2.getType().equals(Type.NUMBER)) {
12      Log.deviantOperatorCallNoNumber(ii);
13    }
14
15    double[] d1 = v1.toDifferential();
16    double[] d2 = v2.toDifferential();
17    double[] r = Factory.differential();
18    r[0] = d1[0] * d2[0];
19    for(int i=1; i<r.length; i++) {
20      r[i] = d1[i]*d2[0] + d1[0]*d2[i];
21    }
22    return Factory.initNumber(r);
23  }
```

4.5 Provided Libraries

A lot of functionality needed in the context of generative modeling is not directly available in Euclides (and often not even in the target language), like vectors or matrices. This functionality is typically provided by means of libraries – and Euclides handles it that way. Euclides offers the following libraries:

- **IO**: The standard Euclides library for input and output offers basic functionality to read and write text and files.
- **MATH**: The standard Euclides library for math provides mathematical functions typically needed in the context of generative modeling.
- **MATH_LA**: Matrices and vectors are available through the standard Euclides library for linear algebra.
- **UTL**: The standard Euclides utilities library is a collection of convenience functions for arrays and parsing of data types.
- **GUI**: Basic functionality to create user interfaces is available in the standard Euclides GUI library.
- **GEO**: The standard Euclides library for geometry offers high-level functionality to control 3D scene (e.g., camera, light, background, ...).
- **GEO_SHAPE**: Basic shapes, as well as the ability to create custom shapes is available in the the standard Euclides GEO_SHAPE library.
- **GEO_MATERIAL**: The standard Euclides library GEO_MATERIAL offers a collection of pre-defined materials.

The two different types of libraries are cross-platform libraries and libraries using native code. While libraries like MATH_LA and GEO_MATERIAL are cross-platform libraries, i.e., libraries not using platform-specific (native) code, other libraries like GUI and IO rely on native code in order to provide their respective functionality.

IO (Input & Output) Due to its platform-specific nature, the standard Euclides IO library is relying on native code. While IO operations can be directly mapped to GML and Java functionality, IO is handled differently in JavaScript. Text IO, in the sense of the Euclides library, is meant to be done via a standard IO channel (`stdin` and `stdout`), but JavaScript offers no feature for that. Franz Papst tackles this problem by using `prompt()` and `alert()` for `stdin` and `stdout`, resulting in a dialogue box every time these functions are called. The browser console (i.e., `console.log()`) is used to output errors.

Reading and writing of files in a browser is done using the API provided by HTML5. However, for security reasons, files are only kept within a sandbox. Files can only be read from the same origin as the website that is trying to access them. In our case we are displaying locally generated Euclides output and thus it is not directly possible to arbitrarily access files on the hard disk. So far no workaround has been implemented to solve this situation other than changing the browser's security options.

MATH The standard Euclides MATH library provides a number of basic mathematical functions (e.g., trigonometric and logarithmic functions). Since these functions are not platform-dependent, it is a cross-platform library.

MATH_LA (Linear Algebra) Vectors and matrices are implemented in the MATH_LA library. Operations like adding, subtracting and multiplying are implemented for both, vectors and matrices. Since these functions are not platform-dependent, it is a cross-platform library.

UTL (Utilities) The standard Euclides utilities library is a collection of convenience functions for arrays (e.g., cloning of arrays or removing of elements) and parsing of data types. Since these functions are not platform-dependent (because they rely on Euclides data types), it is a cross-platform library.

GUI A basic set of functions to create a GUI is available through the standard Euclides GUI library. It is possible to initialize a GUI (i.e., creating a frame, respectively a window, with a title as well as a size defined by width and height). The 2D part with elements like buttons and sliders is initialized by the function `gui_init_2D()`, whereas the function `gui_init_3D()` initializes the 3D part. It is also possible to group standard elements and label them. Any additional styling is not part of the library. Due to the platform-specific nature of GUIs, the standard Euclides GUI library is relying on native code:

- The GML directly offers no possibilities to create standard GUI elements, thus this part of the library is omitted in the GML translator.
- Java offers direct mapping of all GUI functions using the `javax.swing` package and thus fully supports the GUI library.
- In JavaScript, the GUI is created using HTML5 elements. They also offer direct mappings for all functions of the GUI library.

GEO (Geometry) The standard Euclides GEO library enables the creation and manipulation of a 3D scene by means of defining camera, light and other parameters, as well as enabling, disabling and transforming shapes. It also provides functionality for creating shapes and assigning materials or textures to them. Users can create more complex shapes out of a collection of basic shapes, or can directly create (or manipulate) low-level buffers. The buffers are directly used by the GUI library for rendering, or to create output in the form of geometry (OBJ files) and images (PNG files). Some of these functions are platform-dependent (e.g., setting the camera), so the GEO library relies on native code.

GEO_SHAPE (Shapes) The standard Euclides GEO_SHAPE library is a collection of basic shapes (box, sphere, cylinder, ...) and transformation functions. Furthermore, it is possible to create custom shapes by using the function `geo_shape_build(...)`, which takes a number of parameters (buffers and transformation). All functions of this library are not platform-dependent – it is a cross-platform library.

GEO_MATERIAL (Materials) A collection of pre-defined materials, in the form of OpenGL compatible definitions, is available in standard Euclides GEO_MATERIAL library. It can be used to assign materials to shapes, when defining shapes with the GEO_SHAPE library. All functions of this library are not platform-dependent – it is a cross-platform library.

4.6 IDE

The integrated development environment (IDE) of Euclides together with its syntax checker are shown in Figure 4.6.
Like the back-end and the targets, it is written in Java. The integrated editor offers basic functionality for syntax highlighting, font settings, and displaying line numbers to allow for an easy navigation. It is an aim of Euclides to reduce the inhibition threshold in the context of generative modeling. Hence, meaningful error messages are one of the most - if not the most - important aspect of a beginner-friendly development environment. For this aspect, the syntax checker plays an important role. It analyzes the source code and displays detailed messages for warnings and errors in combination with suggestions for resolving issues.

Euclides offers a lot of functionality through libraries (see Section 4.5). Due to missing functionality to declare source code dependencies in ECMAScript 262 (and thus also in Euclides), the IDE offers compiler options for include files. Platform-specific files and libraries can be included by denoting their target.

A total of three compilation targets are available in the IDE: Java, GML, and the documentation target. The Java and GML targets produce executable output, while the documentation target creates an XML-based view of the source-code including variables, functions, statements and files. Settings for the Java target include the definition of the output JAR file, and whether to execute the output using a specific Java Runtime Environment. When compiling for the GML target, the settings are limited to defining the output Extensible GML (XGML) file. The settings for the documentation target offer the ability to

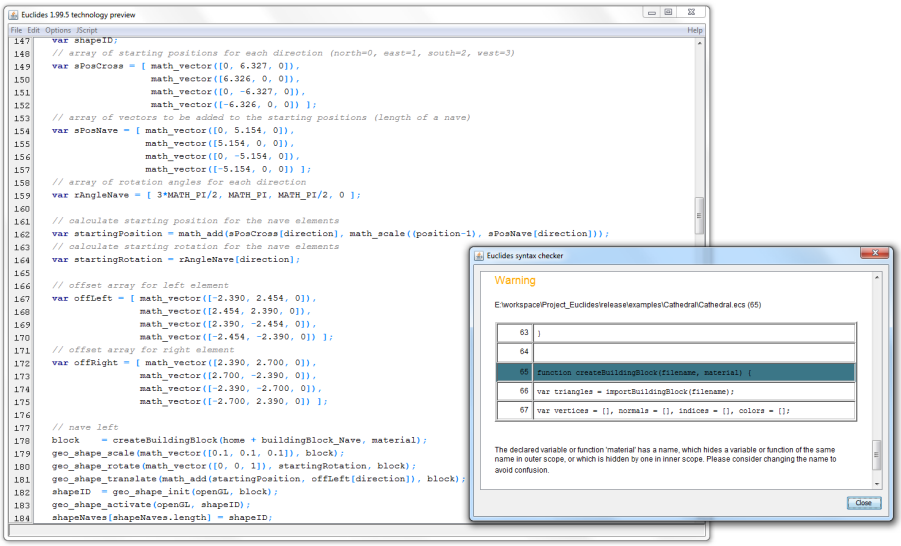


Figure 4.6: The integrated development environment (IDE) consists of a syntax-highlighted editor with line numbers for easy navigation, and a syntax checker with detailed information of warnings and errors.

specify an output directory, and whether to directly open the output in the default browser.

4.7 Interpreter

The simplicity of a programming language is only one factor of a successful development environment. Reasonable feedback and an interactive experience are also important aspects. To offer our users this kind of experience, we enhanced our already existing compiler infrastructure to create an interpreter. In the article "Minimally Invasive Interpreter Construction – How to reuse a compiler to build an interpreter" [SUF12] by Christoph Schinko, Torsten Ullrich and Dieter W. Fellner, we present the interactive interpreter based on the already existing Euclides architecture.

A similar approach to combine interpretation and compilation has been presented by Anton Ertl and David Gregg [EG04], but in contrast to our system, they start with an interpreter and end up with a compiler.

4.7.1 Compilers and Interpreters

Unfortunately, there is no commonly accepted definition of the terms "compiler" and "interpreter". The problem is the smooth transition between compilation and interpretation techniques, which blur a clear distinction. On the one hand many interpreters have integrated just-in-time compilers, on the other hand, some compilers rely on an interpreter integrated into each compiled unit. In combination with virtual machines [LY99], which have functionality not provided by any real machine, and CPUs, which can execute source code directly

[BSK67], it is even more complicated to find a clear distinction.

In the context of Euclides, we differentiate between compiler and interpreter by the number of times the `ASTFactory` (see Section 4.2) is called per JavaScript application execution. If the factory is called every time, the system is called interpreter, otherwise, it's a compiler.

4.7.2 Interpreter Design

In order to design, realize, and implement an interpreter based on an AST, current software engineering approaches recommend one of two main designs: the interpreter pattern and the visitor pattern [HKvdSV11].

According to the interpreter pattern, each node of the AST should have a specialized version of an evaluation, respectively, interpretation method; e.g., `eval(...)`. The visitor pattern in contrast only needs some callback functionality. In this way it can separate algorithms and actions from the data structure it operates on. As the visitor pattern (in combination with an iterator pattern for tree traversal) is already used by the Euclides compiler targets, it is also used by the interpreter.

The main idea of the interpreter implementation is based on a property found in many scripting languages. In contrast to, for example, Java, in which each statement is enclosed (at minimum) by a class definition, enclosed by a file definition, the scripting language JavaScript does not have this "overhead". As a consequence, the root node of the AST is simply a list of statements: `statementA, statementB, statementC` and for each statement, the list of previous statements has to be a valid program. This linguistic property allows to compile each top-level JavaScript statement as a unit of its own – a dynamic library. While this is not sensible for regular compilations, it offers the possibility to compile instructions statement by statement. Finally, if each unit is executed directly after being compiled, the resulting back-end is an interpreter. Even more, additionally included callback routines can be used for debugging purposes [VB07].

4.7.3 Implementation Details

Following the observation that even a single statement can be regarded as a unit of its own, the original JavaScript compiler is extended to reflect this property. Statements in the AST are no longer stored in a one-dimensional array, but a two-dimensional array is used instead. This way it is possible to group statements, i.e., all statements passed to the interpreter in a single evaluation call form one group and are stored in a one-dimensional array. All groups are stored in an array as well, thus as a consequence, the statements are stored in a two-dimensional array. These groups can be accessed by a new set of access functions while at the same time retain compatibility to the compiler, e.g., the command `getAllStatements()` now simply copies the two-dimensional structure in a one-dimensional one.

In addition to the changes in the AST, the namespace is also using a two-dimensional array for storing all symbols the same way the AST does. It uses the same mechanism to create units of symbols while being compatible to the old compiler version. These changes are necessary to allow tracking of interpretation

history as well as to speed up all operations relying on the AST such as validation and code generation.

A small change in the runtime, not related to the interpreter redesign, was carried out in the process of implementing the changes for AST and namespace. Function pointers are now being omitted in the favor of using anonymous inner classes.

4.8 Examples

Based on Euclides' flexible libraries, the creation of models, or visualizations is easily possible. Apart from further use of the translated code on the respective target platforms, the generated geometry can easily be exported to be used in other 3D modeling workflows (e.g., photorealistic rendering). Not only quick examples can be created in just a few lines of code, but also more complex ones. The following examples are translated to the Java target, due to the fact that it is the most advanced and tested target.

4.8.1 Amphitheater

Amphitheaters are open-air venues mostly used for entertainment purposes. Due to their highly regular structure, they represent an ideal use case for generative modeling techniques. The amphitheater model and its visualization (shown in Figure 4.7) have been created in Euclides.

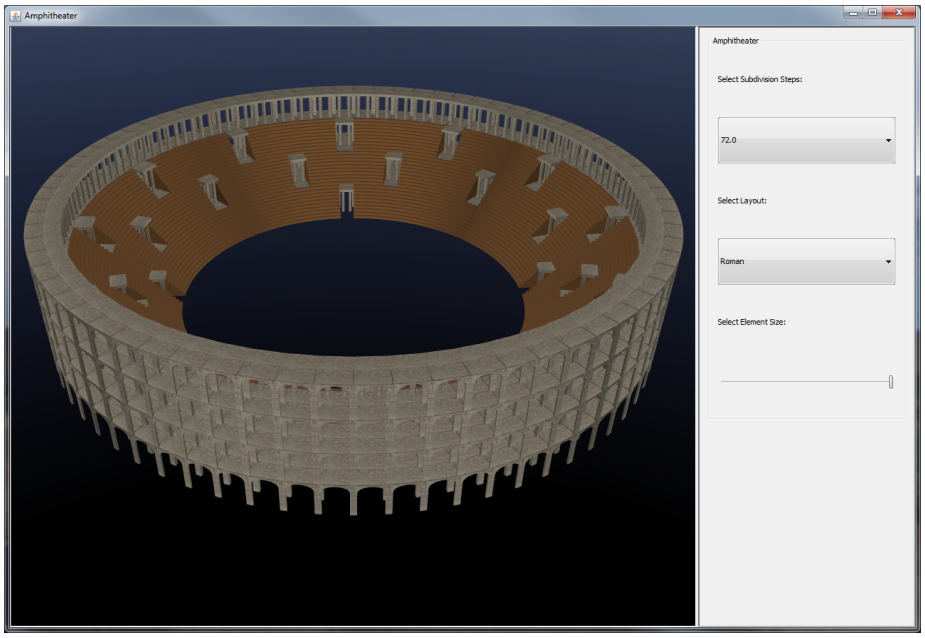


Figure 4.7: The visualization of the amphitheater has been created using Euclides and the Java target. Due to its generative nature and the provided libraries, only 355 lines of code are necessary for the complete example. It is possible to distribute a generative model as an executable JAR file.

The complete example (without libraries) only consists of 355 lines of code. A few high-level parameters are available through the (rather simple) user interface: the size of the structure (a scaling factor), the number of subdivision steps (the number of arcs), as well as different styles of the amphitheater. The whole structure of the amphitheater consists of the primitive building blocks cylinders, boxes and arches with respective materials. The following listing gives an overview of the example and its use of the available libraries (parts of the source code are omitted).

```
1  gui_init("Amphitheater", 800, 600);
2
3  var gui2d = gui_init_2D();
4  var gui3d = gui_init_3D();
5
6  geo_perspective(gui3d, 50.0, 1.0, 0.1, 100.0);
7  geo_camera(gui3d, [0, 0, 10], [0, 0, 0], [0, 1, 0], 1.0);
8  geo_light(gui3d, GEO_LIGHT_OCTAHEDRON, 1.0);
9  var skyTexID = geo_texture(gui3d, "release/examples/Amphitheater
       /images/textures_environment/sky_dayclear_1024.png");
10 geo_skybox(gui3d, skyTexID);
11
12 var SIZE = 0.6;
13 var ROWS = 72;
14
15 function elementStairs(...) {
16    // ... definition of function omitted ...
17 }
18
19 function elementColumn(...) {
20    // ... definition of function omitted ...
21 }
22
23 function elementArch(...) {
24    // ... definition of function omitted ...
25 }
26
27 function main(openGL) {
28    // ... definition of function omitted ...
29 }
30
31 main(gui3d);
32
33 var guigroup1 = gui_group(gui2d, "Amphitheater");
34 gui_label(guigroup1, "Select Subdivision Steps:");
35 gui_combobox(guigroup1, [36, 72, 144], 1, -1);
36 gui_label(guigroup1, "Select Layout:");
37 gui_combobox(guigroup1, ["Roman", "Greek"], 0, -1);
38 gui_label(guigroup1, "Select Element Size:");
39 gui_slider(guigroup1, 0.1, 1, 0.6, -1);
40 gui_label(gui2d, "");
41
42 gui_show();
```

After the GUI is initialized (lines 1-3), perspective, camera, and scene are set up (lines 5-9). The functions `elementStairs(...)`, `elementColumn(...)`, and `elementArc(...)` are used to create a single vertical slice of the amphitheater. These functions directly create geometry based on its parameters. The slices are grouped together and bent (circular, elliptical, ...) in the *main(openGL)* function. The 2D GUI elements (sliders, combo boxes, ...) for interaction with the parameters are defined afterwards (lines 32-39). Finally, the function

`gui_show()` displays the example.

Using the Java target, it is possible to distribute the example as an executable JAR file.

4.8.2 Cathedral Construction Kit

Using more complex building blocks than cylinders or boxes as a basis for modeling reflects the idea of generative modeling. However, these blocks do not necessarily have to be of generative nature. The cathedral construction kit in Figure 4.8 is created in Euclides. It is based on static building blocks and thus offers variability on a rather high level.

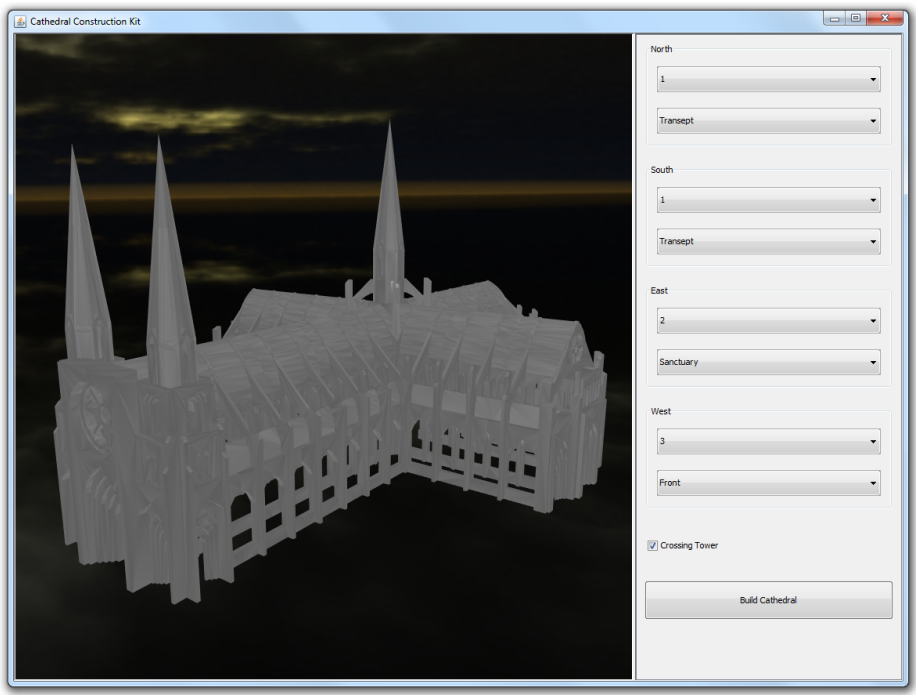


Figure 4.8: The cathedral construction kit has been created using Euclides and the Java target. It offers the possibility to create custom cathedrals based on high-level building blocks.

Its main building blocks have been created by thingiverse[1]. The following building blocks are available: crossing tower base, crossing, front façade left, front façade right, sanctuary left, sanctuary right, nave, tower, transept façade left, transept façade right (see Figure 4.9, and Figure 3.5 in Section 3.3.2). They can be arranged using a few lines of code, or, like in this example, by nine high level parameters, which are exposed in the user interface. With these parameters, it is possible to control the length of transept, sanctuary and the front of the cathedral. Unusual configurations can be created by choosing different terminal building blocks (e.g., a cathedral with three sanctuaries). An optional crossing tower can be toggled on and off. The whole cathedral construction kit (without

[1] http://www.thingiverse.com/thing:2030

libraries) is realized in 456 lines of JavaScript code. Note that in contrast to the amphitheater example, static building blocks are used.

Using the Java target, it is possible to distribute the example as an executable JAR file.

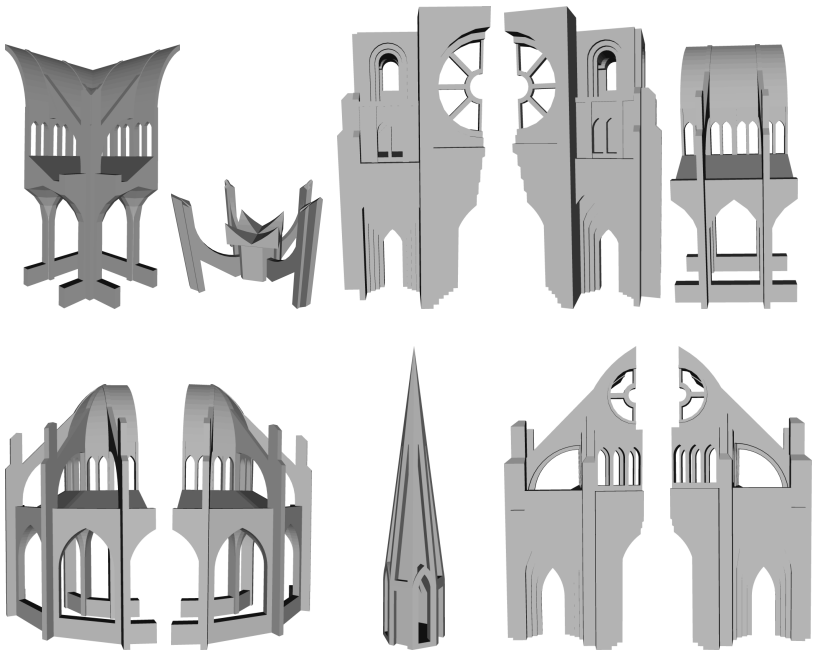


Figure 4.9: The castle construction kit is based on the following ten building blocks (from left to right and top to bottom): crossing tower base, crossing, front façade left, front façade right, sanctuary left, sanctuary right, nave, tower, transept façade left, transept façade right.
The building blocks have been created by Michael Curry, http://www.thingiverse.com/thing:2030

4.8.3 Lorenz Attractor

Another example created in Euclides, can be seen in Figure 4.10. It is a visualization of the Lorenz attractor first studied by Edward N. Lorenz in 1963 [Lor63]. It is a set of chaotic solutions of the Lorenz system which, when visualized, resemble a figure eight.

The appearance of the attractor is controlled by five parameters, which are exposed in the user interface: Prandtl number, Rayleigh number, Beta number, Euler step size, and Euler iterations. In order to visualize its shape, the predefined primitive shape tube is used. The whole example is realized in a little over 100 lines of code, with a mathematical part occupying about half of the program.

Commonly used values, as well as possible values, are defined in the beginning (lines 1-14). The three coupled, non-linear differential equations are evaluated and visualized using the functions `lorenzODE(point)`, `lorenzStep(point)`, and `lorenzEuler()`.

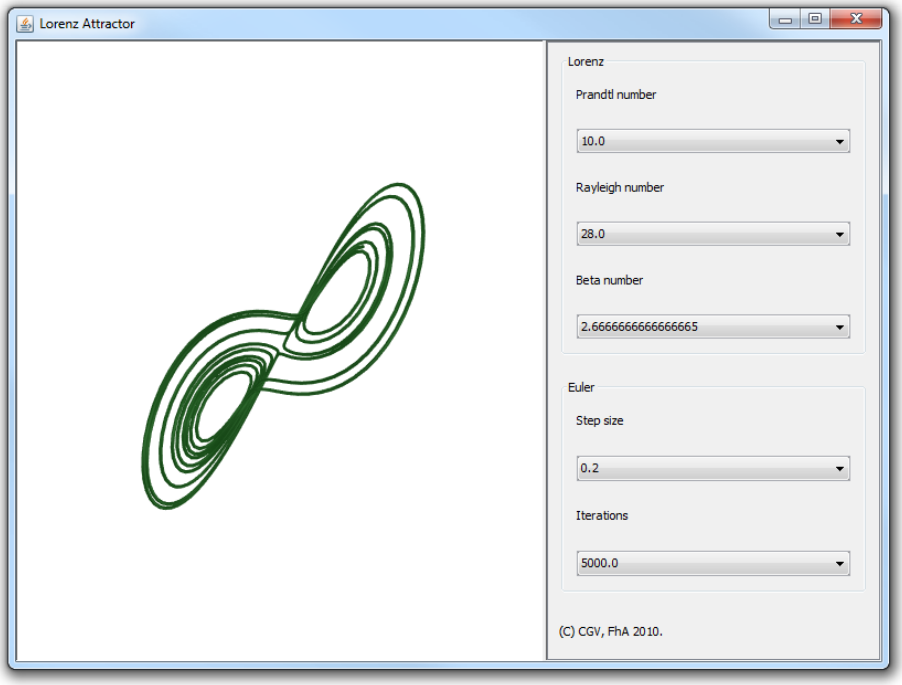


Figure 4.10: The Lorenz attractor (first studied by Edward N. Lorenz) is visualized using Euclides. Five parameters control the appearance of the attractor.

4.9 Summary

This chapter focused on the novel meta-modeling approach called Euclides. It consisted of sections describing the architecture, language elements, different target platforms, provided libraries, the development environment, an interpreter based on the Euclides framework, and a small collection of examples.

Euclides addresses the problem of implementing and maintaining several generative descriptions on different platforms by introducing the concept of generative meta-modeling. A single implementation of all necessary construction rules in an easy-to-use language can be translated to different target platforms. Its high-level representation of the input code allows to preserve the level of abstraction when translating to a target platform. The system creates target code with a clear correspondence to the input code, thus simplifying debugging and reuse.

The presented differential Java target is a basic functionality of the parameter fitting and shape recognition approach. It is also an integral part of other applications in the context of inverse generative modeling. These will be described in the next chapter.

Chapter 5

Inverse Modeling

This chapter presents work based on solving the inverse modeling problem; i.e., what is the best generative description of one or several given instances of an object class?

Related work on the topic is presented in the following section. Afterwards, the parameter fitting and shape recognition technique of Torsten Ullrich is used to analyze digitized objects in terms of changes and damages. A reference surface is constructed using a generative description, whereas the actual object is a laser scan or photogrammetric reconstruction without any additional semantic information. In this context, the meta-modeler approach presented in the previous chapter serves as a basic technology.

Furthermore, this system can be used to perform a template transfer enabling the design of shapes using both low-level details and high-level shape parameters at the same time. This is possible by transferring features from one shape to another.

Contents

5.1 Examples

The inverse problem can be interpreted in different ways. Probably the simplest way is to create a generative model out of a given shape and to store it in a geometry definition file format, as described in Section 2.4.2. A cube can be described as an OBJ file:

```
# cube

# definition of vertices
v -1.0 -1.0  1.0
v  1.0 -1.0  1.0
v -1.0  1.0  1.0
v  1.0  1.0  1.0
v -1.0  1.0 -1.0
v  1.0  1.0 -1.0
v -1.0 -1.0 -1.0
v  1.0 -1.0 -1.0

# definition of faces
f 1 2 3
f 3 2 4
f 3 4 5
f 5 4 6
f 5 6 7
f 7 6 8
f 7 8 1
f 1 8 2
f 2 8 4
f 4 8 6
f 7 1 5
f 5 1 3
```

The notation of this file format can be interpreted as a simple language in Polish prefix notation. Its distinguishing feature is that it places operators (v, f, ...) to the left of their operands or parameters. Since this generative description can only represent a single object, not a family of objects, it is not the desired result.

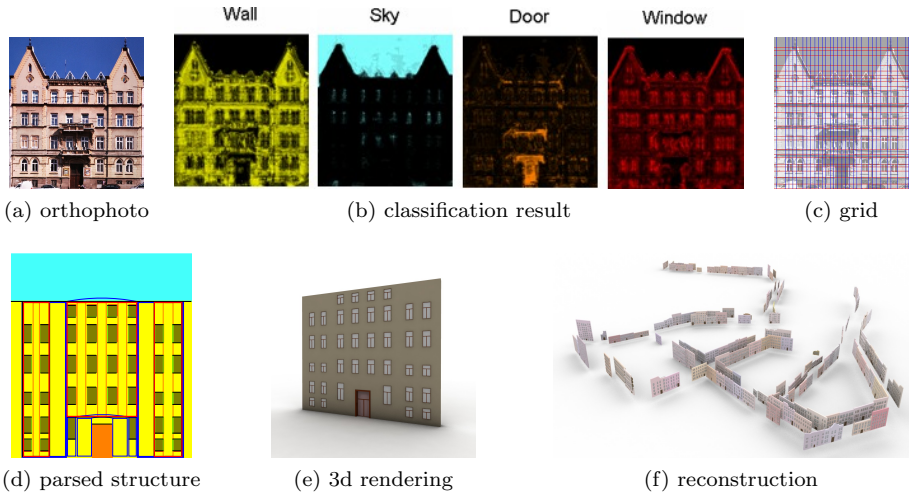
We describe the following fields of application for inverse modeling techniques in the article "A Survey of Algorithmic Shapes" [KSU15] by Ulrich Krispel, Christoph Schinko, and Torsten Ulrich.

5.1.1 Parsing Shape Grammars

Shape grammars can be used to describe the design space of object classes, e.g., of buildings / façades. In the context of inverse modeling, an interesting application is to find an application of rules to describe a given set of measurements of a building. The applied rules can also be seen as parse tree of a given input.

Hayko Riemenschneider et al. [RKT+12] utilize shape grammars to enhance the results of a machine learning classifier that is pre-trained to classify pixels of an orthophoto of a façade into categories like windows, walls, doors and sky. The system applies techniques from formal language parsing to parse a two-dimensional split grammar that consists of horizontal and vertical splits, as well as repetition and symmetry operations. To reduce the search space, an irregular grid is derived from the classifications, and the parsing algorithm is applied to obtain the most probable application of rules that yields a classification

label per grid cell. Such a parse tree can easily be converted into a generative description, as can be seen in Figure 5.1.



(a) orthophoto (b) classification result (c) grid

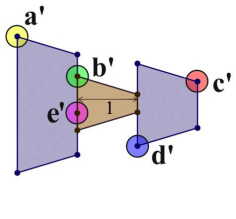
(d) parsed structure (e) 3d rendering (f) reconstruction

Figure 5.1: A building façade (a) is classified into pre-trained categories using a classifier (b). From these classifications, an irregular grid is derived (c), and a two-dimensional split grammar is parsed (d). It can be seen that the system was able to detect repeated columns (red rectangles) and two side parts (connected blue rectangles). The resulting parse tree can be transformed to a generative description, which can be evaluated to geometry (e). Using this technique, parts of the city of Graz were reconstructed from photographs and range scans (f). (Source: Hayko Riemenschneider et al., 2012)

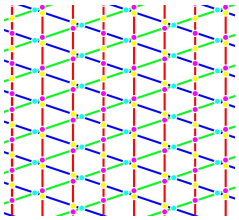
Fuzhang Wu et al. [WYD+14] also address the problem of how to generate a meaningful split grammar explaining a given façade layout. Given a segmented façade image, the system uses an approximate dynamic programming framework to evaluate if a grammar is a meaningful description. Please note that the work does not contribute to the problem of façade image segmentation.

5.1.2 Model Synthesis

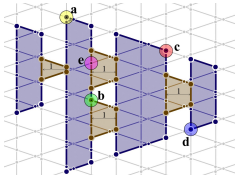
The work of Paul Merell and Dinesh Manocha [MM08] deals with the automatic generation of a variation of shapes. Given an exemplary object and constraints, the task is to derive a locally similar object. The method was inspired by texture synthesis methods. These methods generate a large two-dimensional texture from a small input sample, where the result is locally similar to the input texture, but should not contain visible regularities or repetitions. The method computes a set of acceptable states, according to several types of constraints, and constructs a set of parallel planes that correspond to faces orientations of the input model. Intersections of these planes yield possible vertex positions in the output model. The system proceeds by assigning an acceptable state to a vertex and remove incompatible states in its neighborhood. It terminates, if every vertex has been assigned a state. This process is illustrated in Figure 5.2.



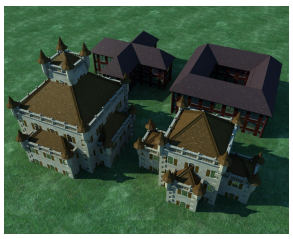
(a) input model



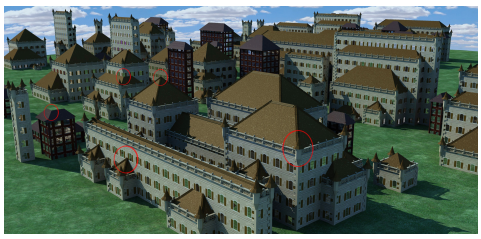
(b) parallel lines



(c) acceptable output shape



(d) input model



(e) generated variation

Figure 5.2: In their work, Paul Merell and Dinesh Manocha [MM08] use a mesh with constraints as input (a) to identify a set of identical lines. Parallel translation of these lines yields a discretization of space (b), from which a new model is synthesized, that locally satisfies the constraints of the input model (c). The bottom row shows an example in 3D, where many complex buildings (e) are generated from four simple ones (d). The output contains vertices that have been constrained to intersect in four faces, some of them are circled in red. (Source: Paul Merell and Dinesh Manocha, 2008)

5.1.3 Inverse Procedural Modeling of Trees

Ondrej Stava et al. [SPK+14] propose a method to estimate the parameters of a stochastic tree model, given polygonal input tree models, such that the stochastic model produces trees similar to the input. Finding such a set of parameters is a complex task. The parameters are estimated using Markov Chain Monte Carlo (MCMC) optimization techniques. The method uses a statistical growth model that consists of 24 geometrical and environmental parameters. The authors propose a similarity measure between the statistical model and a given input mesh that consists of three parts:

- shape distance, which measures the overall shape discrepancy,
- geometric distance, which reflects the statistics of geometry of its branches, and
- structural distance, which encodes the cost of transforming a graph representation of the statistical tree model into a graph representation of the input tree model.

For some examples see Figure 5.3. The MCMC method has also been applied to find parameters of a statistical generative model: [TLL+11], [VGDA+12], [YYT+11].

(a) procedural tree model

(b) input model and variations generated from the procedural model

Figure 5.3: The method of Ondrej Stava et al. [SPK+14] uses a statistical growth model of trees that is able to generate a variety of different tree species, as can be seen in the top row (a). The system performs a statistical optimization to find the parameters of the model, given an input exemplar. The bottom row (b) shows an input model and three variations generated from the procedural model using the parameters that resulted from the optimization process.
(Source: Ondrej Stava et al., 2014)

5.1.4 Parameter Fitting and Shape Recognition

The approach presented by Torsten Ullrich and Dieter W. Fellner uses the generative modeling system presented in Chapter 4 to describe a class of objects and to identify objects in real-world data e.g., laser scans [UF11], [USF08], [SUSF11]. The input data sets of the algorithm are a point cloud P and a generative model M. Then, the algorithm answers the questions

1. whether the point cloud can be described by the generative model and if so,

2. what are the input parameters x_0 such that $M(x_0)$ is a good description of P.

The implementation uses a hierarchical optimization routine based on fuzzy geometry. It simply regards a generative script as a function M with input parameters $x \in \mathbb{R}^k$.

The example data set shown in Figure 5.4 consists of laser-scanned cups and a generative cup description. It takes 15 parameters: (x, y, z) is the base point of the cup and (α, β, γ) define its orientation. Its shape is defined by an inner $f_{in}(x) = \frac{1}{25}(x + \frac{1}{10})^{2+shape}$ and outer $f_{out}(x) = x^{3+shape}$ shape function with one free parameter *shape*. These functions are rotated around the cup's main axis and scaled with the parameters r and h. The handle is defined via six parameters forming points in 2D (in the plane of the handle, which is defined implicitly by its orientation γ); namely $(h_1, f_{out}(h1))$, (h_{2A}, h_{2B}), (h_{3A}, h_{3B}),

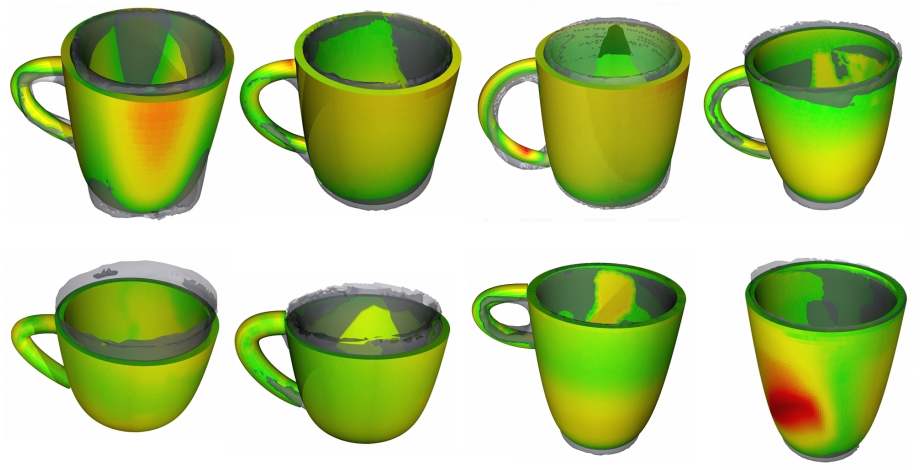


Figure 5.4: The scanned cup (rendered in semi-transparent gray) have been identified as instances of the generative cup description. The color codes the geometric distance from the generative instances to the scanned cups (green: close, red: more deviation). In these cases the cups' properties (position, orientation, radius, height, handle shape) are determined successfully.
(Source: Torsten Ullrich, 2011)

and $(h_4, f_{out}(h_4))$. They are the control points of a Bézier curve. Its tube with a fixed diameter (10mm) defines the cup's handle.

The algorithm is able to detect an instance of the generative cup. In these cases the cups' properties (position, orientation, radius, height, handle shape) are determined with a small error.

5.2 Real-World Comparison

The parameter fitting and shape recognition technique described in Section 5.1.4 can be used to analyze digitized objects in terms of changes and damages. In the articles "Procedural Descriptions for Analyzing Digitized Artifacts" [USSF13] by Torsten Ullrich et al., "Real-World Geometry and Generative Knowledge" [SSUF11a] by Thomas Schiffer et al., and "Variance Analysis and Comparison in Computer-Aided Design" [SUSF11] by Christoph Schinko et al., we propose a work flow, which automatically combines generative descriptions with reconstructed objects and performs a nominal/actual value comparison. The reference surface is a generative description whose accuracy and systematics describe the semantic properties of an object, whereas the actual object is a real-world data set (laser scan or photogrammetric reconstruction) without any additional semantic information. These analysis and documentation tasks are needed not only in the context of cultural heritage but also in engineering and manufacturing.

5.2.1 Architecture

The system consists of three main parts: registration, analysis, and visualization (see Figure 5.5).

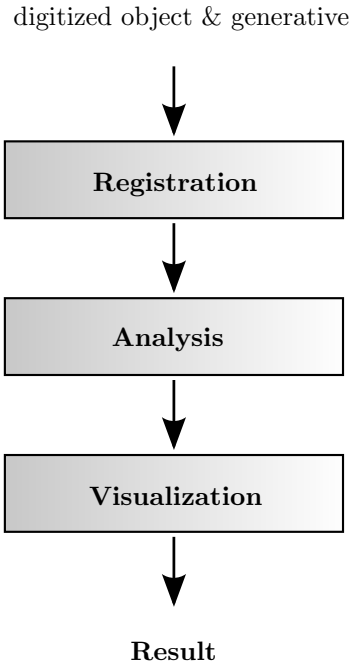


Figure 5.5: The architecture of the system consists of three parts: registration, analysis, and visualization. A generative model is registered against a digitized object. The difference between them is calculated and finally visualized.

Registration The registration part fits, respectively registers, a generative model, which can be regarded as a function M with parameters x, to the real-world data set (see Section 5.1.4). The registration step takes M and determines a parameter set $x0$, such that $M(x0)$ fits a given object S.

Analysis The analysis part computes the difference between the generative model $M(x_0)$ and the object S using state-of-the-art ray tracing techniques.

Visualization The results of the analysis are visualized using X3D technology. An X3D file is generated containing the generative model $M(x_0)$, a texture of distance values and shader code for applying the difference as displacements. This allows selective switching between the two models or displaying both of them simultaneously.

5.2.2 Registration

The processing pipeline starts with the registration step. During this step a generative model with its free parameters is registered (fitted) against an input data set. As example, we use a digital object of the *Museum Eggenberg* collection and a corresponding generative description shown in Figure 5.6. It shows a digitized object (left) and a generative description (right), which will be registered to each other in a first step.

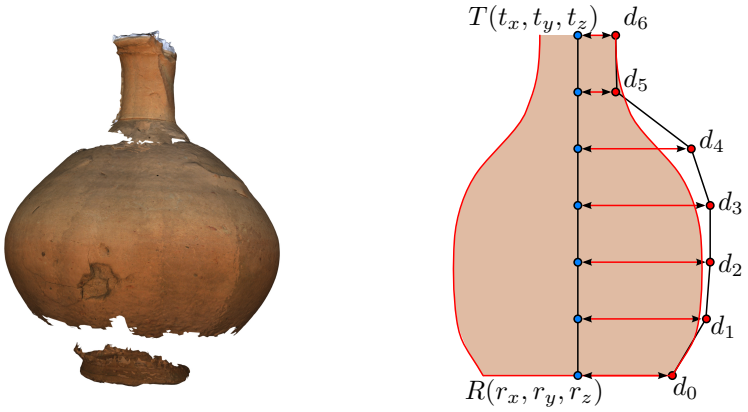


Figure 5.6: A digitized object is often represented by a list of triangles with additional information such as textures (to describe the visual appearance) and meta data (to describe its context); e.g., the vase on the left hand side is a digitized object of the *Museum Eggenberg* collection. It consists of 364 774 vertices and 727 898 triangles.

The generative model to describe a vase takes 13 parameters: $R(r_x, r_y, r_z)$ is the base reference point of the vase in 3D and $T(t_x, t_y, t_z)$ is its top-most point. The points R and T define an axis of rotational symmetry. The remaining seven parameters define the distances $d_0, \ldots, d_6$ of equally distributed Bézier vertices to the axis of rotation. The resulting 2D Bézier curve defines a surface of revolution – the generative vase.

The registration step regards a generative script as a function with input parameters. These parameters may have a semantic meaning (width, height, etc.). Each set of parameters describes a member of the family of objects defined by the generative description. For example, changing the parameters d_0, ..., d_6 of the generative vase shown in Figure 5.6 (right) creates different variations of vases as illustrated in Figure 5.7.

The registration estimates these free parameters; i.e., it determines the best-fit parameter vector $p^\star(r_x^\star, r_y^\star, r_z^\star, \quad t_x^\star, t_y^\star, t_z^\star, \quad d_0^\star, d_1^\star, d_2^\star, d_3^\star, d_4^\star, d_5^\star, d_6^\star)$.
This step is performed using numerical analysis techniques to minimize the distance between the geometry of the digitized object on the one hand side and the geometry of a member of a generative family [Ull11].

A numerical optimization evaluates the generative script up to several thousand times. The Differential Java target (see Section 4.4.5) is used to perform the gradient-based optimization routines.

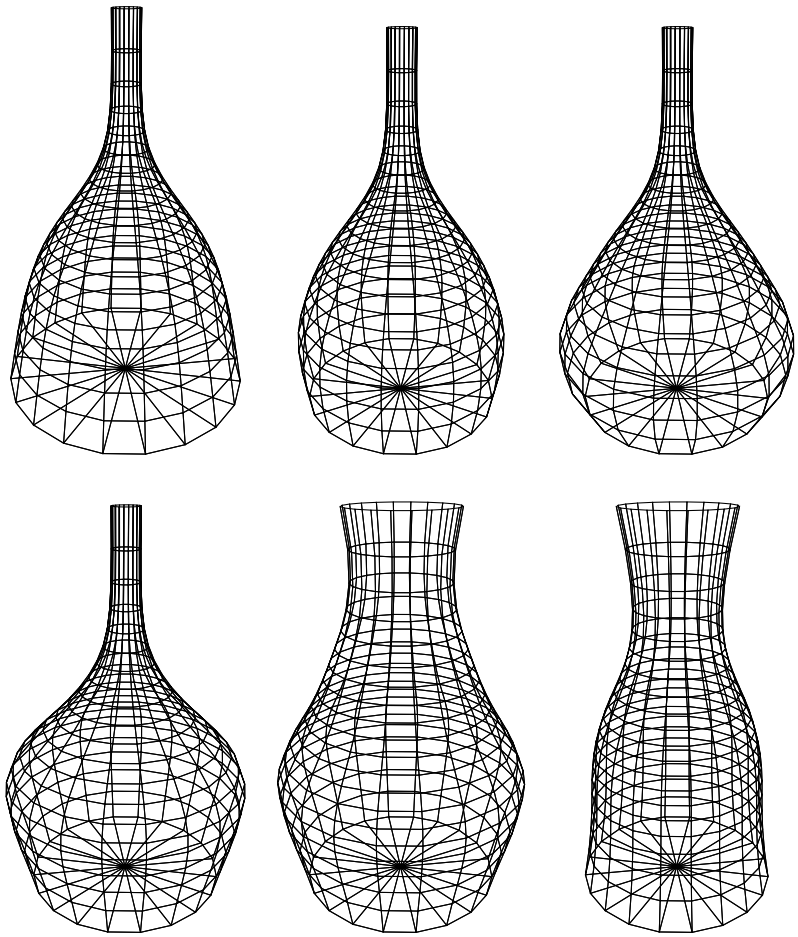


Figure 5.7: The generative vase is defined by two reference points of an axis of rotation (top and bottom) and seven distances d_0, dots, d_6, which define a surface of revolution. Changing only the parameters d_0, dots, d_6, keeps the vases' height fixed and modifies the outer shape.

The result of the registration step is a particular generative vase out of a family of vases. It is the best generative vase to describe the digitized object it has been registered to. Figure 5.8 illustrates this result. Please note, the registration algorithm can only modify the input parameters of the generative description. It cannot modify the description itself. As a consequence, features, which cannot be generated by the script, cannot be included in the generative result.

5.2.3 Analysis

State-of-the-art ray tracing techniques are used to compute the difference between generative and digitized geometry. For this distance computation, we need to intersect a large amount of rays with the static triangles obtained from the geometry acquisition step as described in Section 5.2.3. The ray tracing

Figure 5.8: The digitized object (left) has been registered to a generative description of a vase. The result is the best-fitting vase of the complete family of vases defined by the generative description – i.e., the generative vase (right; rendered in red) is very similar to the digitized object (right; rendered in beige).

library is composed of acceleration structure construction and efficient parallel ray tracing.

Acceleration Structure Construction

To accelerate the ray intersection queries, we organize the triangulated, static geometry in a particular data structure. In a first step, we enclose each triangle with an axis-aligned bounding box (AABB). By grouping them recursively, a tree-like structure is obtained. This kind of hierarchy is called bounding volume hierarchy (BVH) and is a very popular acceleration structure for ray tracing [WMG+07]. The BVH construction can be performed fully automatically by partitioning the geometry according to a cost function and updating the bounding volume of each node. The recursive, top-down algorithm is outlined in the following listing.

```
1  procedure BuildBVH(BVHNode node)
2    if needSplit() then
3      splitGeometry()
4      BuildBVH(LeftChild)
5      BuildBVH(RightChild)
6      computeAABB(LeftChild, RightChild)
7    else
8      computeAABB(objects)
9    end if
10 end procedure
```

We use a binary bounding volume hierarchy BVH, which means that each node has a maximum of two child nodes (called left and right child). The construction is performed recursively and starts at the root node of the BVH, which initially

contains all triangles of the input mesh. If the geometry belonging to a node needs to be split, it gets partitioned among the child nodes, where construction proceeds recursively. Consecutively, each node's axis-aligned bounding box has to be updated, either combining the AABBs of its children or the contained triangles (if no split has been performed).

In our implementation, a node is split whenever it contains more than four triangles. Splitting itself is guided by the commonly used surface area heuristic (SAH) cost function.

$$Prob(ray\,hits\,C \,|\, ray\,hits\,P) \propto \frac{surfaceArea(C.AABB)}{surfaceArea(P.AABB)}. \tag{5.1}$$

It states that, assuming uniformly distributed rays, the probability of a ray intersecting a child node C given that its parent node P is intersected is proportional to the ratio of the surface areas of the bounding boxes. We use a formula proposed in [Wal07], which is based on the SAH, to guide our object partitioning step.

$$cost(L, R) = N_L A_L + N_R A_R \tag{5.2}$$

N_L and N_R represent the number of primitives and A_L and A_R the surface area of the AABB for a given partition $L \uplus R$ of geometric objects. This function tries to minimize the overall intersection costs of a ray with the BVH by grouping suitable AABBs. To reduce the search space for partitions, the centroids of the AABBs are projected along each coordinate axis and sorted in ascending order. The partition with minimal costs can then be found by "sweeping" a split plane along the axis (see also [WMG+07]). This is done for all three coordinate axes selecting the split with minimal costs.

Parallel Ray Tracing

A previously constructed BVH can be traversed in depth-first order. The recursive approach visits all nodes that are intersected by a ray in strict front-to-back order. To maintain this node ordering, each ray requires a stack to store some intersected nodes for later use. The algorithm is outlined in the following listing.

```
1  procedure IntersectBVH(Ray ray, BVHNode node)
2    if node is a leaf then
3      intersectTriangles(ray)
4    else
5      t_left = intersectAABB(LeftChild, ray)
6      t_right = intersectAABB(RightChild, ray)
7      if t_left and t_right then
8        if t_left < t_right then
9          pushNode(RightChild)
10         IntersectBVH(ray, LeftChild)
11       else
12         pushNode(LeftChild)
13         IntersectBVH(ray, RightChild)
14       end if
15     else
16       if t_left then
17         IntersectBVH(ray, LeftChild)
18       else if t_right then
19         IntersectBVH(ray, RightChild)
20       else
```

```
21          node = popNode()
22          if(node)
23            IntersectBVH(ray, node)
24          end if
25        end if
26      end if
27    end if
28  end procedure
```

The traversal is started from the root node using the ray as parameter. If the node is a leaf, the contained geometry is intersected with the ray and the ray parameters are updated. An inner node has two child nodes, which are both intersected with the ray. If both children are intersected, the child that is closer to the ray's origin is traversed recursively first, while the farther one is pushed onto the stack. If just one of the children is intersected, traversal proceeds to this child immediately. When no child node was intersected, a node is popped from the stack and traversal continues. The intersection search stops, if the stack gets empty and no more nodes are left for traversal. This depth-first order traversal can be used to find a ray intersection in $O(\log(n))$ time on average, instead of the naive $O(n)$, where n denotes the number of triangles.

To handle millions of intersection queries in parallel, we implemented the depth-first traversal on the GPU using NVidia's CUDA technology [LNOM08], [Nvi17c]. As mentioned before, the constructed BVH is transferred to GPU memory, where it is subsequently traversed in massively parallel fashion. We use the approach of Timo Aila and Samuli Laine, which maps one ray to one thread [AL09]. All threads execute our implementation of the presented algorithm to find their intersection points.

Distance Calculation and Encoding

The ray casting library is embedded into the distance calculation, which takes the generative reference model from the registration step as well as the real-world model as input data sets. Both models are now static geometry, as the registration only returns the best-fit solution and not the complete family of models. Once the mesh data is available, the ray casting library is initialized with the real-world model.

The main idea is now to calculate to offset between generative reference model and digitized object and to store it in the reference model's texture. In this way both models are combined in one data set. The generative model describes the nominal/ideal surface whereas the geometric offset in its texture stores the real-world model.

The first step is to find sample points and corresponding normals on the registration model to obtain rays used for distance calculation. A trivial approach would be to just use the vertices and normals defined by the supplied mesh. However, this approach may lead to a rather coarse sampling resolution. Texture coordinates seldom degenerate to one single (u,v) coordinate for all vertices defining a primitive – may it be a triangle or a quad. Therefore it is meaningful to find a finer sampling resolution for the distance calculation. This can be achieved by taking into account the texture resolution of the generative model.

Simple and Efficient Normal Encoding

Compressing normal information can be achieved by texture- or vector-based algorithms. Texture compression algorithms are image-based approaches, which use *uv*-space coherence for efficiency. Vector compression algorithms only regard a single (normal) vector at a time without any context. An overview of compression techniques can be found in the article "Fast normal vector compression with bounded error" [GKP07].

In the article "Simple and Efficient Normal Encoding with Error Bounds" [SUF11] by Christoph Schinko, Torsten Ullrich and Dieter W. Fellner, we regard the problem of normal vector compression as an optimization problem. It answers the question "How can n points be distributed on a unit sphere such that they maximize the minimum distance between any pair of points?" [Wei17]. This maximum distance is called the *covering radius*, and the configuration is called a *spherical code*. With $n = 2^b$, the points represent an optimal encoding of normalized vectors in b bits.

For a resolution of b bits, 2^b normals on a unit sphere can be represented. Our approach subdivides the unit sphere into six congruent sides with a regular square pattern on each side. We generate $m = 6 \times s \times s$ points with $s = \left\lfloor \sqrt{2^b/6} \right\rfloor$. All points are numbered consecutively, and only a point's ordinal numeral is stored. This scheme has a cutting loss, but due to its regularity, the compression and decompression step can be performed with a fixed number of arithmetic operations.

As the projection of a bounding box grid onto a sphere results in unequally distributed patches concerning their size, we use a spherical, angle-based parameterization, which delivers much better results. Figure 5.9 visualizes the spherical codes.

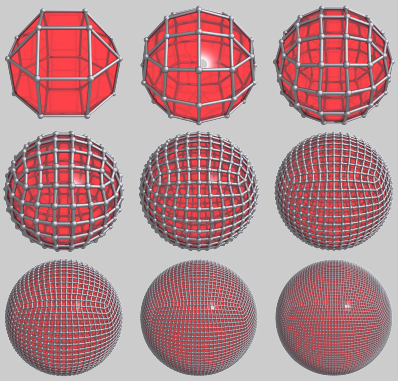


Figure 5.9: The compression scheme can represent only a limited number of normals for a fixed resolution. These configurations are visualized using 24 vertices (5 bits, top-left), and up to 7776 vertices (13 bits, bottom-right).

The results can be measured in angular distance between any pair of points. Table 5.1 shows the values for different resolutions with corresponding error bounds.

bits	normals	point distances	
		avg.	err. lim.
4	6	90.00	45.00
8	216	13.08	6.541
12	4 056	3.030	1.515
16	64 896	0.760	0.380
20	1 048 344	0.189	0.095

Table 5.1: The algorithm generates spherical codes depending on the given resolution (bits). Each configuration consists of a fixed number of normals, whose distance to each other (measured in degree) is limited (error limit).

The idea is to not only use the texels associated with the vertices defining the primitives, but to use all texels inside a primitive. These additional sampling points have unique corresponding texel values to be used to store distance values. To fill these texels with distance values it is necessary to obtain their position and normal direction in object coordinates. This is an easy task at the border texels of the primitive – since these values are available directly, but involves calculating object coordinates out of texel values for all other texels.

With all texels and their the corresponding object coordinates on the generative reference model, these coordinates together with two vectors (in positive and in negative normal direction) represent rays to be tested against the scanned model. The ray casting library calculates the intersection points – if there are any – up to a predefined maximum distance.

These distance values are encoded into an image/texture, following a predefined scheme. We store the distance of a hit in positive and negative direction as unsigned byte in the red, respectively green channel of the texture. The blue channel is used to encode the available distance information:

- A value of 0 means that there is no hit in positive or negative direction.
- If there is a hit in negative direction, the value of 64 is added.
- In case there is a hit in positive direction, the value of 128 is added.

A schematic illustration of a distance texture is shown in Figure 5.10 (left).

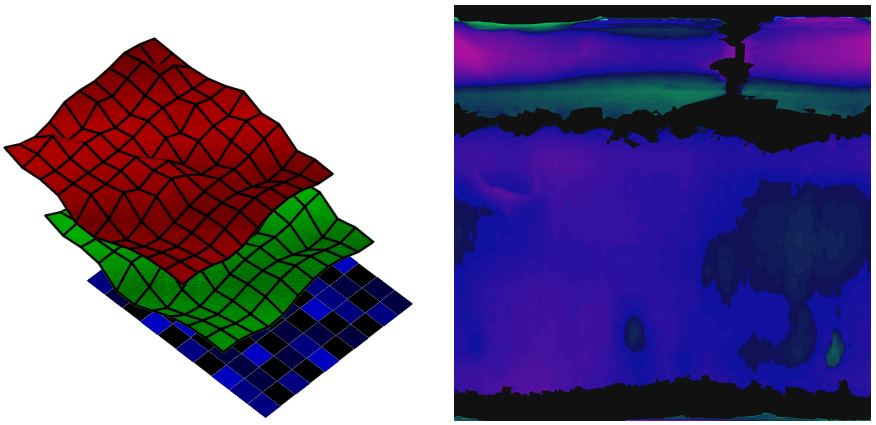


Figure 5.10: The distance texture (right) of our example vase is used in a displacement shader to create an output mesh, which ideally matches the target mesh. Three color channels are used to encode distance values (left). The red and green channels are used to store distance values, whereas the blue channel is used to store additional information. Please note, contrast and brightness have been adjusted for illustration purposes.

It shows the three channels RGB in a slightly shifted manner. Furthermore, the distance channels red (positive direction) and green (negative direction) are visualized as heightfields. Furthermore, Figure 5.10 (right) shows the distance texture of our example vase: high distance values are encoded in brighter colors,

low distance values are encoded in darker colors, missing parts are encoded in black.

The texture encoding allows to carry out a selective displacement in the geometry shader incorporated in the output X3D file. A built-in exporter creates the X3D file.

5.2.4 Visualization

The last step combines the generative model, the input data set and its offset description in an X3D file including shader code capable of applying the difference as displacements. This allows switching selectively between the two models or displaying both of them simultaneously. The generated X3D file incorporates the following components:

- the geometry of the generative, best-fit reference mesh as indexed face set,
- a texture storing distance values to the digitized object,
- as well as vertex, geometry and fragment shaders for displacement and lighting purposes.

The reference mesh and the distance texture are the input for the shader stages. The vertex shader primarily acts as a pass-through stage. Position, normal and texture coordinates of the input vertex are handed over to the geometry shader for further processing. The geometry shader accepts triangles as input and output type. For each input vertex, a texture lookup reveals whether the vertex needs to be displaced. In case there is a distance value in the texture, the displacement of the three vertices is calculated and an output triangle is generated. When distance values in positive and negative direction are available, two triangles are emitted. In all other cases there is no output at all. Additionally, the shader allows to recursively subdivide input triangles to meet the desired output resolution of one vertex for almost every texel. This way we obtain a good representation of the digitized object. As a last stage, *Blinn-Phong* lighting is carried out on a per pixel basis in the fragment shader.

5.2.5 Examples

To demonstrate the workflow, we use an example data set, which consists of vases and pottery of the *Museum Eggenberg* collection. A selection of the objects is shown in Figure 5.11. These objects have been reconstructed using the photogrammetric service Arc3D[1], which returns triangle meshes.

The registration and optimization routines take one of the input data sets (see Figure 5.11) and the generative description and automatically determines the best-fit parameter vector. The next step performs the analysis and stores its result in a texture map. The per-texel-distances are encoded using the RGB channels of an image.

During the optimization process and the analysis step further details are determined automatically:

- direct shape parameters such as height and radius,

[1] http://www.arc3d.be

Figure 5.11: The example data sets are photogrammetric reconstructions of objects of the *Museum Eggenberg* collection. They consist of 344 516 (left), 364 774 (middle), 405 838 (right) vertices and 687 767 (left), 727 898 (middle), 810 749 (right) faces respectively triangles.

- derived shape parameters such as volume,

- analysis values such as maximum deviation and irregularity (assuming the generative model can represent the digitized object adequately).

Figure 5.12 visualizes some results. Each rendering shows the clean generative model (light red) for parts, which do not have a counterpart in the input data set, and the geometric offset (light brown) for parts, which do have a corresponding part in the input data set.

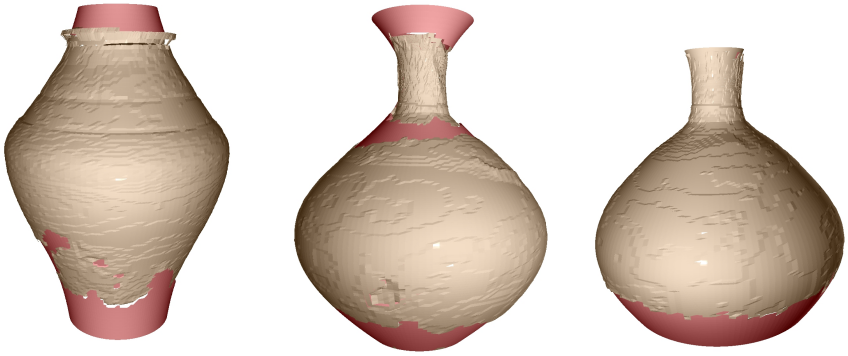


Figure 5.12: Using the pipeline presented in this article, digitized objects and their corresponding generative descriptions can be combined automatically and visualized interactively. The generative parts (light red) are drawn only if no corresponding counterpart in the input data set (light brown) exists.

5.3 Shape Modeling

The pipeline to analyze digitized objects described in Section 5.1.4 can also be used to modify existing 3D shapes. In the article "Modeling with High-Level Descriptions and Low-Level Details" [SUF14] by Christoph Schinko, Torsten Ullrich and Dieter Fellner, we show that high-level descriptions can be used to resemble real-world objects or create new ones. In this way, we can design shapes using both low-level details and high-level shape parameters at the same time.

Similar to the architecture described in Figure 5.5, the system relies on registration, analysis and visualization. The additional template transfer step enables the generation of new shapes by altering the parameters of the generative description (see Figure 5.13).

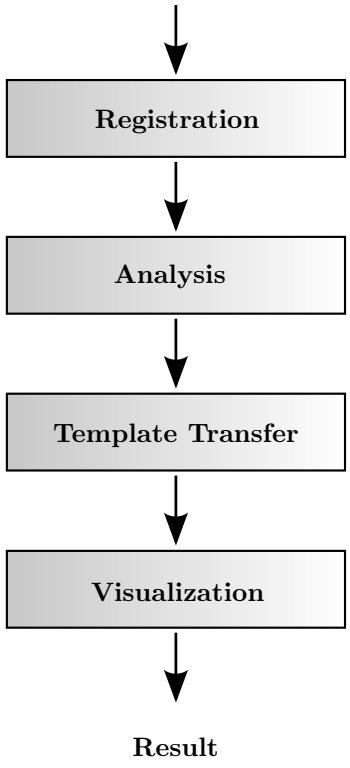
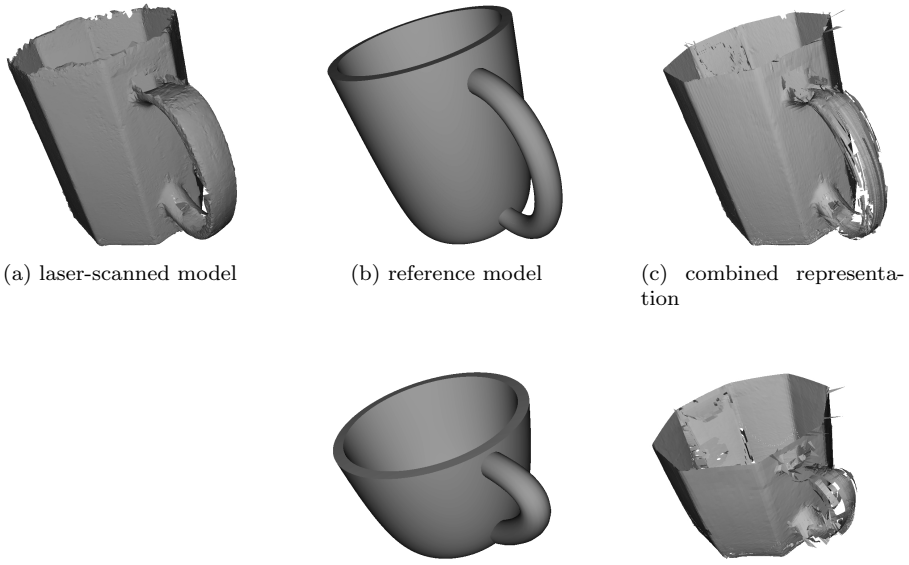


Figure 5.13: The architecture of the system consists of four main parts: registration, analysis, template transfer, and visualization. A generative model is registered (fitted) against a digitized object. The difference between them is calculated and can be used to generate new shapes.

To demonstrate this workflow, we use a laser-scanned cup and the generative cup description from Section 5.1.4. The laser scan of the cup has 66243 vertices and 130973 faces (triangles). Because the real cup has a clean and glossy surface, it has been difficult to scan. The scan result is noisy and it is not cleaned-up (i.e., mesh repairing, hole filling, mesh smoothing has not been done) has many holes (see Figure 5.14 (a)). The registration and optimization routines of the shape recognition determine the best-fit parameters automatically. The result is shown in Figure 5.14 (b). The combined representation is shown in Figure 5.14 (c).



(a) laser-scanned model (b) reference model (c) combined representation

(d) new model (e) transferred combined representation

Figure 5.14: This figure shows the scanned model (a), the procedural reference model (b), as well as the output of the combined representation (c). The combined version consists of a static instance of the reference model. It has been defined by a set of parameters obtained in the fitting process. Having modified the parameters of the generative description, new procedural cups can be generated (d). If one of these new cups is combined with an already existing texture, previously captured details can be transferred (e).

As the geometric offset is applied to the generative model on-the-fly, it can be exchanged as long as the texture coordinates are generated consistently. In this way, it is possible to modify the cup's high-level parameters (height, radius, ...) while its low-level details are preserved. For demonstration purposes we designed a Cappuccino cup; i.e., we chose appropriate parameters of the generative model (a smaller height h and radius r and a slightly different shape parameter). Having replaced the "big mug" by our new Cappuccino cup, we obtained an octagonally shaped version. Figure 5.14 (d, e) shows the generated Cappuccino cup and its noisy, octagonally shaped version.

5.4 Summary

The beginning of this chapter focused on introducing inverse generative modeling. The section about related work was concluded with the parameter fitting and shape recognition technique of Torsten Ullrich relying on the differential Java target of the Euclides meta-modeling approach presented in the previous chapter.

This technique was subsequently used to analyze digitized objects in terms of changes and damages. Since a generative description resembling an ideal object often lacks wear and tear effects of real-world objects, the fine details of real-world objects were therefore applied onto the generative description. Thereby, deficiencies of the real-world object can be identified and analyzed.

The system can also perform a template transfer from one object to another. Having a generative description with applied offset at hand, it is possible to create new objects by changing the initial parameters of the generative description. This enables the design of objects using both low-level details and high-level parameters at the same time.

After covering different modeling paradigms, a final step in the shape processing pipeline discussed within this thesis is concerned with proper visualization of 3D content. The next chapter will focus on this aspect describing two visualization techniques tailored towards specific use cases.

Chapter 6

Visualization

In order for humans to perceive and interact with shapes, a vital part of shape processing is visualization. For that purpose, computer graphics provides means to create, manipulate, and interact with the different representations, especially within a virtual world, either in virtual reality or in a simulation environment. Depending on the use case, specialized software and hardware may be required. A scientific visualization of an abstract scientific process has different requirements regarding, for example, degree of realism, interactivity, or expressiveness than video games. This not only has an impact on the choice of shape representation, but also on the visualization hardware. While work from previous chapters focuses on describing and modeling shapes (a necessary prerequisite), the focus of the following chapter lies on visualization techniques.

The first section presents a system to seamlessly project onto non-planar surfaces. It is used to create dynamic, reconfigurable spaces influencing the behavior of groups and individuals. In contrast to the rather artistic use-case of this projection system, an immersive, autostereoscopic visualization system for a driving simulator is presented in the next chapter. Both systems heavily differ in their requirements showing the diverse application fields for visualization.

Apart from virtual representations, models can also be physically created using 3D printing devices. Julian Gardan gives an overview of current additive manufacturing technologies [Gar16].

Contents

6.1 Non-Planar Projections

An exciting idea for future visual installations is to use a deforming space as "passe-partout" for stunning responsive audiovisual experiences. Dynamic, reconfigurable spaces are an exciting possibility to influence the behavior of groups and individuals. They may have the potential of stimulating various different social interactions and behaviors in a user-adapted fashion. Projecting on larger surfaces can be a problem for a single projector. Problems might be the limited light intensity of the projector, or limited distance to the projection surface. Composing a projection of multiple projectors solves these problems at the cost of a more complex visualization setup. While this works well on a static surface, the tiling becomes visible when the surface dynamically moves and deforms.
The use of tiled displays on non-planar screens has become feasible through the availability of high-resolution cameras for calibration. The idea of the calibration is to allow for geometric alignment of the projectors by capturing and analyzing structured patterns projected onto the unknown, deformed, projection surface. In order to deal with moving and deforming display surfaces, the calibration process must allow for real-time operation. To achieve this goal, it is desirable to process the geometric correction only on a single captured image.

The calibration of dynamically deforming surfaces resembles shape reconstruction. Some techniques, like stereo algorithms, can be used in both cases (for example, by Daniel Scharstein and Richard Szeliski [SS03], or by Andreas Griesser and Luc Van Gool [GG06]), or point cloud reconstruction (e.g., by Patrick Quirk et al. [QJS+06]).

When using a tiled projection display, projectors are used both for geometric correction and to display content. This can be done beforehand, in case of a non-deforming projection surface. In order to accomplish both display and measurement simultaneously, imperceptible structured light techniques were developed. Ramesh Raskar et al. present a combination of multiplexing and light cancellation techniques to hide patterns in a series of white-light projections with a synchronized camera [RWC+98]. With this method, seamless panoramic display systems with multiple cameras and projectors can be built [RBY+99]. However, there is a trade-off between updating the patterns in such a frequency, that changes in the display geometry can be captured fast enough, and not disturbing the content projection too much.

The method proposed by Daniel Cotting et al. embeds arbitrary binary patterns into images displayed by unmodified Digital Light Processing (DLP) projectors [CNGF04], [CZGF05]. Images encoded in that way are visible only to the cameras synchronized with the projectors.

The continuous display surface auto-calibration method by Ruigang Yang and Greg Welch does not require dedicated hardware. It uses a camera to observe the display and match image features with the corresponding features that appear on the display [YW01]. This method relies on corresponding geometric features in consecutive frames, which highly depends on the content.

Jin Zhou et al. present a continuous self-calibration method for planar surfaces using a camera attached to a projector [ZWAY08]. Necessary re-scanning in case the surface geometry is changed, disturbs the content projection.

The method proposed by Johnny C. Lee embeds a small set of optical sensors in the display screen to discover the projection area in combination with

the projection of a series of gray-coded binary patterns [LDMA+04]. However, this technique works only for front projections, since the sensors are embedded in the screen.

Despite much research and development in the context of tiled projections, several challenges remain. In most of the existing work, well-calibrated projectors and sensors in a permanently installed environment are used. In practice, a large tiled projection system is typically needed only temporarily, e.g., for short term events like stage performances and exhibitions. In these case, a reduction of the required hardware and calibration effort is an important aspect. Another aspect is the robustness and adaptability of the calibration. In a rented exhibition location, equipment, such as stage lighting or the sound system, is stationary and hardly movable. This imposes special requirements on hardware and software.

In the articles "Tiled Projection Onto Bent Screens Using Multi-Projectors" [KSHF13] by Hyosun Kim et al., as well as "Tiled Projection Onto Deforming Screens" [KSH+15] by Hyosun Kim et al., we present a quick and efficient method to project a coherent, seamless and perspectively corrected image from one particular viewpoint using an arbitrary number of projectors. The method has been developed in the context of a project called *Responsive Open Space* initiated by *ORTLOS Space Engineering Graz/London*[1].

6.1.1 Exhibition Setup

At the location *Dom im Berg* (Graz, Austria), a large projection surface (the size of 8×6 m) was suspended between electric telescope cylinders (see Figure 6.1).

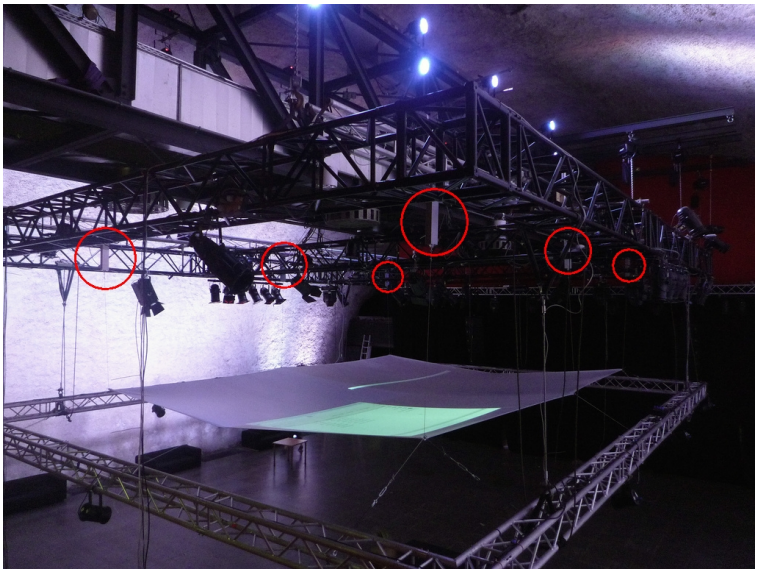


Figure 6.1: The projection setup consists of four projectors projecting onto a movable fabric used as projection surface. It is suspended between electronically controlled actuators (marked by red circles) that can move vertically.

[1] http://www.ortlos.com/space_engineering/

An interactive visualization was triggered by visitors interacting under the projection surface – the interaction was captured by a *Microsoft Kinect* sensor. A total of four projectors, connected to a single computer, were used for rear-projection visuals from a team of creative artists who participated in the project. The projection surface was heavily deformed by its own weight, and the movement of the electric telescope cylinders.

6.1.2 Reconfigurable Projection Geometry

An overview of the framework consisting of offline preparation, online adaptation, corrected projection, and user interaction is shown in Figure 6.2.

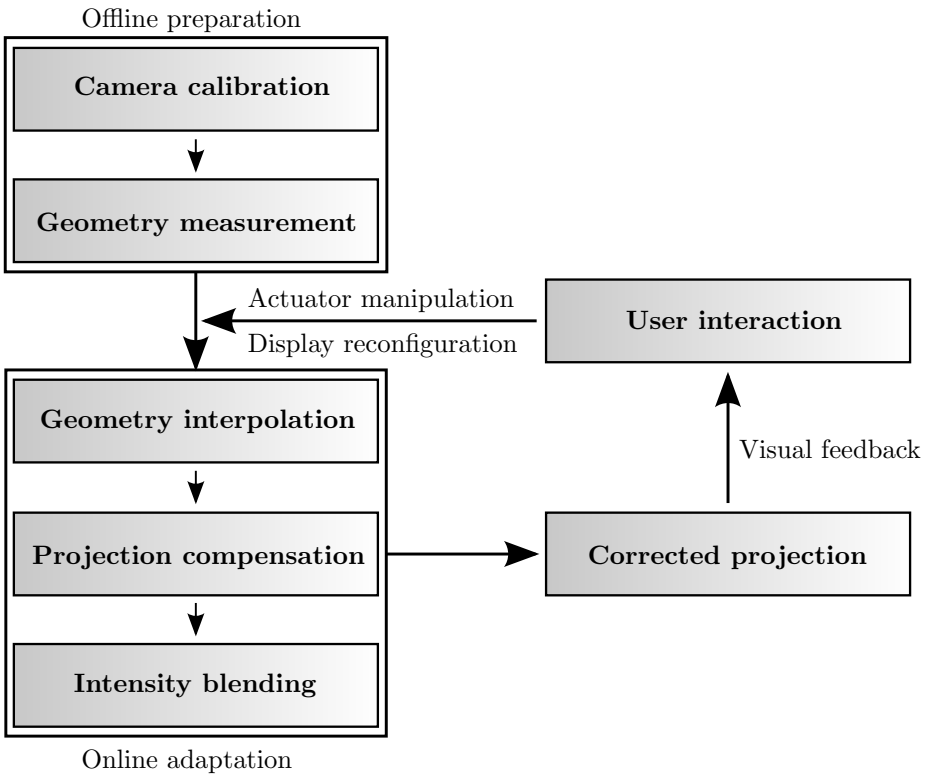


Figure 6.2: The framework consists of offline preparation, online adaptation, corrected projection, and user interaction.

After calibrating the camera, the offline phase proceeds by projecting a sequence of checkerboard patterns on the display surface. The actuators are in an initial position. Pattern recognition is performed for each actuator configuration, to extract the grid positions of the feature points in the image captured for each projector. When using fewer configurations, more interpolation has to take place between the number of key positions. The produced checkerboard grid describes the geometric shape distortion of the display surface with respect to

the configuration. The grid can be directly used as (u, v) texture coordinates to undistort the projected images.

In the online adaptation, the tiled images of the multi-projector display must appear (from a specific point of view) as if they were produced by a single projector. The procedure to determine and compensate the geometric relationship between the overlapping projectors is the key component of the framework and consists of three parts (see Figure 6.2):

- Estimating the surface deformation
- Compensating the deformations
- Blending the image intensities (soft edge blending)

Offline Preparation

The geometry of the display surface can be estimated using pixel references between the projected image and the camera image. By using a checkerboard pattern, the geometric registration considers the surface as an arrangement of piece-wise planar tiles, since the feature points are exactly located at the corners of the tiles. The estimation of the surface using a checkerboard pattern depends on the complexity (frequency) of the surface, thus a checkerboard with a certain density is required. A suitable checkerboard density is chosen manually.

The robust checkerboard recognition method consists of two steps: intra-corner point detection and outermost corner point estimation (see Figure 6.3 on the left).

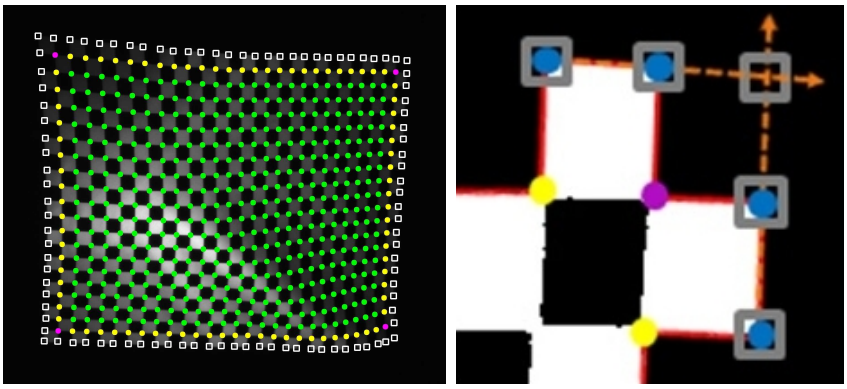


Figure 6.3: A quad mesh is generated out of the detected intra-corner points. Three classes of corners with different numbers of intra-neighbors are marked by pink (two neighbors), yellow (three neighbors) and green (four neighbors) dots (left). Two vectors passing through the convex hull points (blue) intersect where the corner of the checkerboard pattern is likely to be located (right).

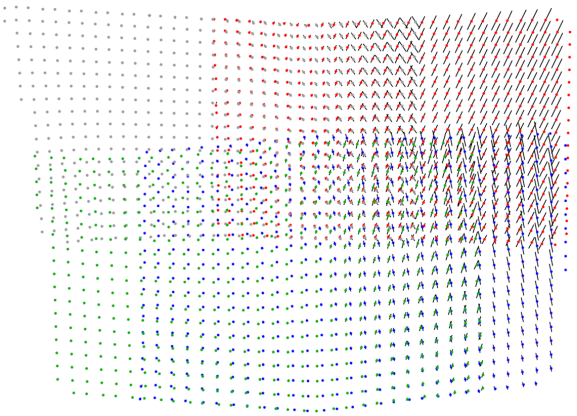
Intra-corner points are approximated by using the method proposed by Weibin Sun et al. [SYXH08]. Adjacent candidate pixels with four alternating dark and bright neighbors are merged into clusters using connected component labeling. The cluster centers are extracted and refined through an iterative process. After

all intra-corners are detected, the pre-defined checkerboard pattern geometry (i.e., its size) is used to match the grid from the potentially large number of results that were found.

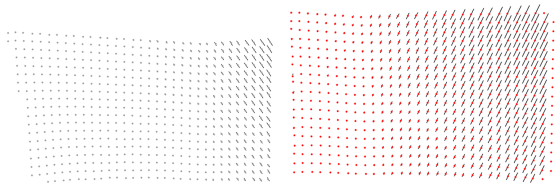
Detecting corner points on the boundary of checkerboard pattern requires a different approach due to vanishing contours of dark tiles. The contour around the pattern boundary is generated to extract the convex hull as a set of candidate outermost corner points. Two vectors passing through the convex hull points intersect where the corner of the checkerboard pattern is likely to be located (see 6.3 on the right).

Geometry Interpolation

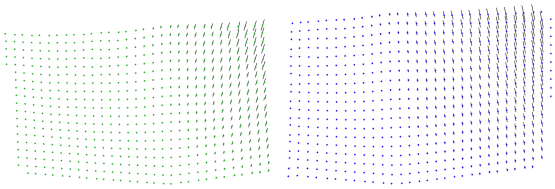
A detailed view of the measured grid data is illustrated in Figure 6.4.



(a) combined view



(b) top left projection (c) top right projection



(d) bottom left projection (e) bottom right projection

Figure 6.4: A combined view (a) of the linear grid variance shows the configuration $(1, 0, 1, t, 1, 0)$. By reducing t (i.e., moving one of the actuators), the grid points move towards the top right corner. The dots represent the grids for $t = 0.5$, whereas the black line segments denote the connection between the values $t = 0$ and $t = 1$. Viewing the individual grids (b,c,d,e) reveals that the points travel linearly on their segments.

The effect on the grid points when varying a single rod parameter shows a linear correlation. The shape of the flexible surface is determined by six parameters that can be seen as a six-dimensional vector, e.g., $(1, 0, 1, 0, 1, 0) \in [0, 1]^6$. By varying one parameter, e.g., $(1, 0, 1, t, 1, 0)$ for $t \in [0, 1]$, it turns out that the grid points travel linearly between the two grids for $t = 0$ and $t = 1$. Figure 6.4 (b,c,d,e) shows the points for $t = 0.5$ lying in the middle of their respective line segments. Despite the fact that the shape of the surface is deformed in a nonlinear way, the grid points move along a straight line. The reason for this linearity are the light rays of the projector. Figure 6.5 shows that the intersections of the deformed, moving surface with the light rays of the projector P all lie on a straight line.

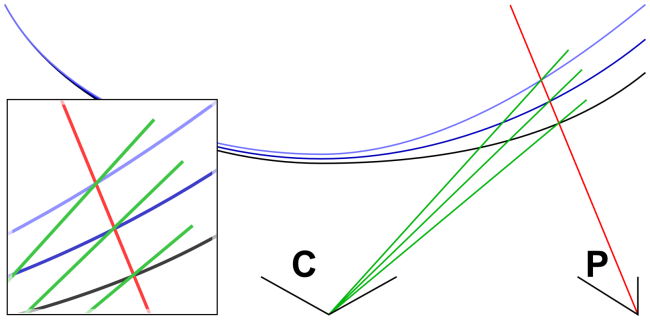


Figure 6.5: A ray from projector P (red) intersects the deforming surface on a straight line; locally, the displacement is linear with respect to the displacement of the rod, which explains the linearity observed by the camera C (green).

When seen from the camera C, this linearity is preserved. Although each of the six parameters pulls the grids in different directions, all grid points move linearly with respect to the variation of any single parameter. A specific configuration $(t_1, t_2, t_3, t_4, t_5, t_6)$ can be obtained by six successive changes:

$$(0, 0, 0, 0, 0, 0) \to (t_1, 0, 0, 0, 0, 0) \to \ldots \to (t_1, t_2, t_3, t_4, t_5, t_6)$$

Due to the fact that each change is linear, and that they can be carried out in any order, they must be the result of multi-linear – in this case hexalinear – interpolation.

To obtain the function value for a given parameter vector, linear interpolation along the first coordinate yields 32 interpolants, then 16 along the 2^{nd}, 8 along the 3^{rd}, 4 along the 4^{th}, 2 along the 5^{th}, and finally one along the direction of the last coordinate. A total of 63 linear interpolations are needed. Even thought the interpolation has to be performed between grids, this can be easily accomplished in real time.

Projection Compensation

Projection compensation is carried out in the programming language and development environment Processing to be able to easily incorporate visualizations (scripts) from creative artists [RF07]. For each projector, the texture coordinates from the respective grid are handed over to the Processing script for displaying a part of the viewport. The viewport is defined in pixel coordinates

of the camera. A framebuffer object in the size of the viewport is used for off-screen rendering of the visualizations. The framebuffer object is then used as a texture for a pre-defined planar quad mesh whose vertex positions are the previously calculated texture coordinates. This step compensates the deformation of the projection surface from the viewpoint of the camera, as illustrated in Figure 6.6.

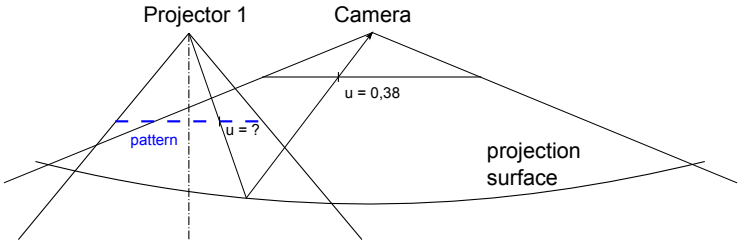


Figure 6.6: A camera acquires the checkerboard pattern that is projected on the screen. For each grid point of the pattern, the u-coordinates (in the one-dimensional example) can be measured in the camera image. These coordinates can be directly used as texture coordinates to generate an undistorted projection from the viewpoint of the camera.

Intensity Blending

Since the projections have to overlap in order to cover the whole screen, these areas look brighter than the areas covered by a single projector. To obtain a consistent brightness over the whole surface, the pixel intensities in the overlap regions have to be attenuated using alpha blending.

The overlap area can be exactly determined from the grids. The pixel attenuation is proportional to the shortest distance to the boundary of the overlapping region. The sum of the attenuation rates at corresponding projector pixels has to sum up to one. Then the alpha weight $a(p_{u,v})$ at a given pixel $p_{u,v}$ can be calculated using the function

$$a(p_{u,v}) = (1 - \beta)^{\frac{1}{\gamma}}$$

where β is the attenuation rate and γ is the adjustment factor accounting for projector gamma (see Figure 6.7).
The alpha map for each projector is generated using the inverse mapping of the distance map to the boundary of the overlap regions. Note that for obtaining seamless color transitions a radiometric calibration of the projectors would have to be performed.

6.1.3 User Interaction

The test setup incorporates user interaction (i.e., movement under the projection surface) in the form of displacement of the screen. Movement is captured by a Microsoft Kinect sensor. The transfer of this data into movement of the projection surface is done by a 3$^{\text{rd}}$ party software. The software provides the input data to control the position of the actuators, e.g., to lower the screen above a detected group of persons. After establishing a corrected projection

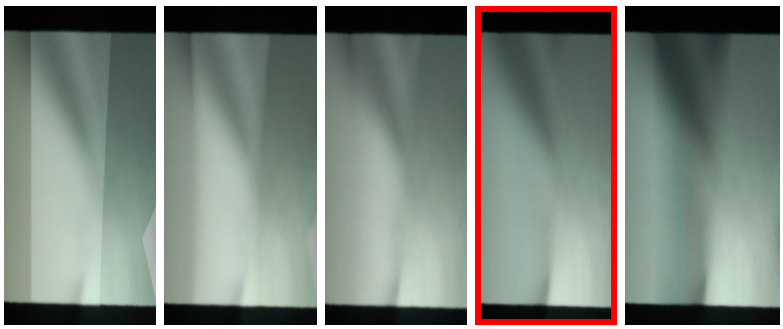


Figure 6.7: Choosing the appropriate projector gamma is essential. From left to right: No blending, blending with $\gamma = 0.1$, $\gamma = 0.3$, $\gamma = 0.6$, $\gamma = 0.8$. The correct choice in this case is $\gamma = 0.6$.

setup, the positions of the actuators serve as input for the online adaptation of the projection environment.

6.1.4 Test Setup and Results

To test and improve the accuracy of our system, and because of limited availability for testing with the projection setup at the event, we built a scaled-down version of the suspended screen with a size of 1.2×0.90 meters (see Figure 6.8).

Figure 6.8: The test setup consists of a suspended sheet held in place by six rubber tapes that can be adjusted manually (top part). Vertical displacement of the scaled-down setup corresponds to two meters of the exhibition setup (see Figure 6.1). The four projectors and the wide-angle camera are located below the sheet (bottom part)

In contrast to the setup at the event, we used front-facing projectors, a front facing camera, and manual modification of the suspended projection surface with rubber bands. The camera is a ten megapixel industrial camera (3840 × 2748) with a wide angle lens.

To obtain reliable data for validating the online projection compensation, and to evaluate the accuracy of the estimation, we have measured all combinations of the values $t_{1...6} \in \{0, 1/2, 1\}$, 729 in total. Due to the setup with four projectors, 2916 images had to be processed by the checkerboard extraction. This took about 40 seconds per image, adding up to less than three minutes per configuration. Only $2^6 \cdot 4 = 256$ grids resulting from the 64 extreme positions were used for the online interpolation. The other grids were used for validation. The 2D distance between the interpolated and the measured grid points is only about 3-4 pixels in average, which is about 0.1 percent of the camera image resolution. This error is hardly noticeable, as shown in Figure 6.9.

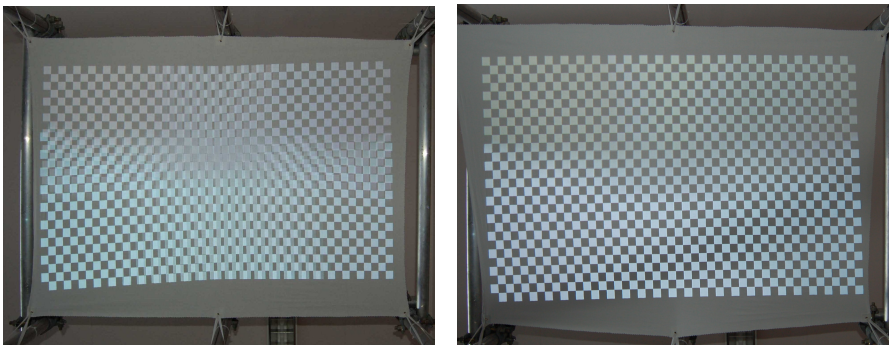


Figure 6.9: Geometric correction of overlapping images removes severe projection artifacts (left) using the hexalinear interpolation of 64 measured grids (right).

Our method is effective in avoiding severe projection artifacts by dynamically adjusting the tiled projection to the deforming surface.

The work on non-planar projections is tailored towards dynamic visual installations. A visualization system for a different scenario – a driving simulator – is presented in the next section.

6.2 Parallax Barrier Displays

A parallax barrier can be used to create an autostereoscopic display, i.e., a display creating stereoscopic images without the need to use special headgear. It is a device placed in front of a display to create stereoscopic, or multiscopic images. While stereoscopy enables the binocular perception of depth, multiscopy displays multiple angles at once, allowing a viewer to see the content from different angles – not just a left-eye / right-eye angle.

Parallax barriers can be created in different ways. It can be a material with a set of precision slits, or a translucent material with an opaque pattern. Despite the chosen realization, it allows each eye to see a different set of pixels. When a labeling is available for every pixel (i.e., seen by left eye, seen by right eye,

seen by both eyes, not seen), it is possible to synthesize a stereoscopic image consisting of pixels taken from renderings of two viewpoints. The functional principle of a parallax barrier is illustrated in Figure 6.10.

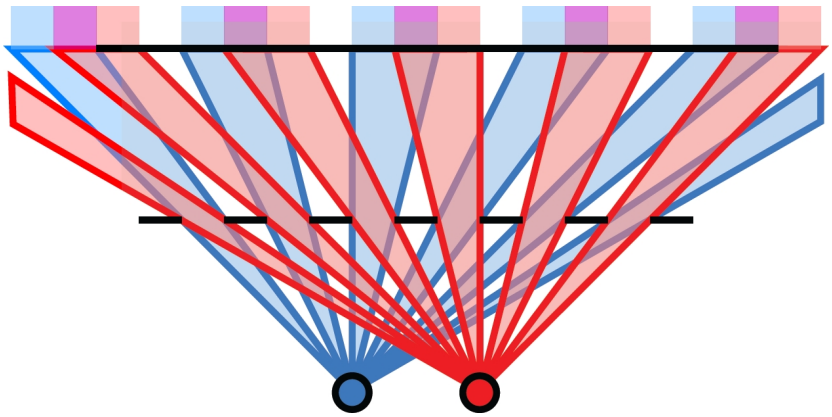


Figure 6.10: The parallax barrier is located between the eyes (visualized in blue and red) and the pixel array of the display. Opaque parts block certain pixels for each eye, while others are visible for one eye only. This results in the eyes seeing only disjoint pixel columns (at least in an optimal setting). If the display is fed with correct image data, the user sees a stereoscopic image.

The principle of the parallax barrier was independently invented more than hundred years ago by Auguste Berthier and Frederic Eugene Ives. Almost a century later, in the early 1990s, Sharp[2] and Dimension Technologies (DTI)[3] (amongst others) developed products using this technology. Sharp was briefly selling laptops and mobile phones with 3D LCD screens [JMW+03], while DTI was selling lightweight backlights for LCDs that generate a special illumination pattern to create autostereoscopic 3D images [Eic98].

Recent developments show that the technology is harder to apply for larger screens (like television sets), because of the requirement for a wider range of possible viewing angles. It is easier to apply for smaller displays, like the Nintendo 3DS hand-held game console and various smart phones [BWS+07]. Apart from its use for 3D displays, the technology can be applied in a different area: to enable a single screen to display two different views at the same time. These dual-view displays are, for example, used for the entertainment system in the 2010-model Range Rover allowing driver and passenger to view different content.

Parallax barrier displays have the limitation that a user's position needs to be fixed or at least constrained to a few viewing spots for the effect to work. This drawback can be eliminated by virtually adjusting the pixel columns such that the separation remains intact, as presented by Daniel J. Sandin et al. [SMD+01] and by Tom Peterka et al. [PKG+07]. To accurately adjust the pixels, some kind of user tracking is needed, which also limits the number of possible users. In order to eliminate the need for user tracking, most consumer products rely on parallax barriers with lenticular lenses. This approach does not put any constraints on the number of possible users at a time, but is relatively

[2] http://www.sharp-world.com/
[3] http://dti3d.com/

restricted concerning the possible viewing position(s).

Additional drawbacks of parallax barrier displays are the loss of display resolution (down to about a half), and the loss of brightness due to the barrier itself. Douglas Lanman et al. presented an approach for content-adaptive parallax barriers to overcome this issues by using an adaptive mask in a dual-stacked LCD [LHKR10]. However, the real-time application for dynamic content in a driving simulator is ruled out due to the involved computational complexity.

6.2.1 Driving Simulator

In the article "Building a Driving Simulator with Parallax Barrier Displays" [SPH+16] by Christoph Schinko et al., we present an optimized stereoscopic display based on parallax barriers for a driving simulator to create an immersive virtual environment. The driving simulator is built around a modified MINI Countryman (R60) chassis. Eight liquid-crystal displays (LCDs) are mounted around windscreen and front side windows. Four 55inch LCDs are placed radially around the hood of the car in a slanted angle. Four 23inch LCDs are used for the two front side windows. The four LCDs in the front are equipped with parallax barriers made of 2cm thick acrylic glass to minimize strain caused by the slanted angle. Each acrylic glass (the parallax barrier) is printed with a custom-made striped pattern, which is the result of an optimization process. The displays are connected to a cluster of four computers with powerful graphics cards. In order to account for movement of the driver's head inside the car, an eye-tracking system from SmartEye[4] consisting of two cameras with infrared flashes is installed on the dashboard of the car. One camera is mounted on the left a-pillar, the other camera is hidden in a ventilation nozzle of the center console. The position-depended rendering of the simulation scenario is performed on the cluster using the InstantReality framework[5].

During the design of the driving simulator, the display configuration has been optimized for maximum coverage of the driver's field of view, in a two-step procedure.

Display Arrangement

The first optimization step determines the best position of each display taking into account the chassis of the MINI Countryman. Furthermore, the displays should cover as much of the driver's field of view as possible, in order to avoid distracting parts of the non-simulation environment being visible. An additional constraint is the maximum angle of inclination of the LCDs in order to function properly.

The optimization uses a cylindrical rendering as a cost function, with the cameras placed at the position of the driver's eyes. Occluding geometry of the vehicle's chassis is supplied with a black, non-visible material, the displays are black as well. The whole scene is placed in a surrounding, illuminated sphere in cyan and yellow – for each eye separately. The geometric setting is illustrated in Figure 6.11 (left), while the result of the optimization is shown in Figure 6.11 (right). The cost function can be minimized easily: Each non-black pixel is

[4] http://www.smarteye.se
[5] http://www.instantreality.org

a disturbing view past the displays and should be removed. Minimizing the number of such pixels improves the result.

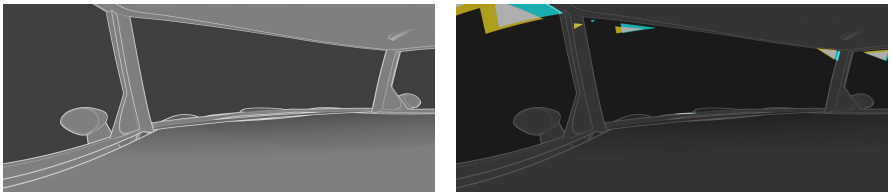


Figure 6.11: In order to optimize the display arrangement, their visibility has been analyzed using cylindrical renderings from the driver's position (left). The result of the optimization routine (right) consists of the occluding vehicle geometry and displays (rendered in black), and the environment (rendered in cyan and yellow for each eye separately). For ease of understanding the geometry (left) is overlaid semi-transparently.

Pattern Optimization

The second optimization step uses the display position and the driver's position in order to optimize the barrier pattern (i.e., the pattern printed on the acrylic glass). In our setup, the barrier pattern is a set of lines described by three parameters; i.e., the line width l_w, the distance between two consecutive lines l_d measured between medial axes, and their deviation (angle) from a vertical alignment l_a. The cost function of this optimization routine depends on these three parameters. In each evaluation, a geometric scene is generated (including the vehicle, the displays, and the line of the barrier pattern). For each scene, the intersections of rays, starting from the position of the driver's eyes to each display pixel, check the visibility of a display pixel. In other words, an intersection test is performed, for all scene, for both eyes, for all pixels. The result for each pixel can be:

- visible by both eyes,
- visible by the left eye only,
- visible by the right eye only,
- not visible at all.

The objective function simply counts the number of pixels, which can be seen by both eyes, or which cannot be seen at all (i.e., which are not separable). Figure 6.12 shows plots for all four front displays seen from a driver's perspective: outer left (top left), inner left (top right), inner right (bottom left) and outer right (bottom right).

The line distance in millimeters between two consecutive lines l_d measured between medial axes is plotted on the x-axis, while the line width l_w is plotted on the y-axis as percentage of the line distance. A fixed angle measured from a zero degree vertical orientation of 10 degrees for the left displays, and -10 degrees for the right displays has been chosen empirically (if not too "extreme", the angle

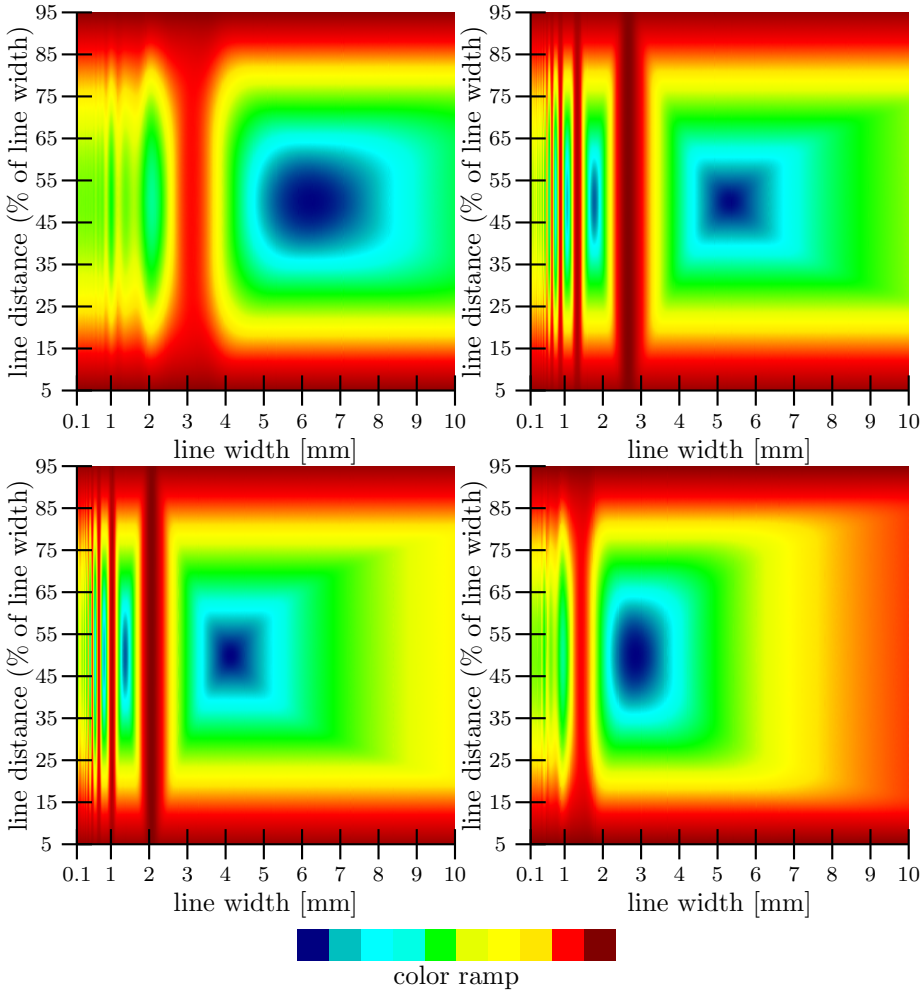


Figure 6.12: The optimization routine plots the cost function for the selected range of barrier parameters for the outer left (top left), inner left (top right), inner right (bottom left) and outer right (bottom right) displays. The line distance l_d in millimeters is plotted on the x-axis, while the line width is plotted on the y-axis as percentage of the line distance. For these plots, the angle is at a fixed value of 10 degrees for the left displays, and -10 degrees for the right ones. The color encodes the separability of the pixels (see color ramp at the bottom). Blue denotes that 100 percent of all pixels are separable. Red denotes that 0 percent of all pixels are separable.

was found to have almost no influence). The color encodes the separability of the pixels (see color ramp at the bottom). Blue denotes a 100 percent separability of the pixels. Red denotes that 0 percent of all pixels are separable. It can be seen that a "sweet spot" exists for all four displays.

While the optimization returns the optimal configuration, the manufactured driving simulator differs significantly from this specification:

- As the displays and the barriers are arranged in an inclined plane, sagging effects in the order of millimeters occur.
- The parallax barrier pattern has not been printed with the required precision; that is, the lines are up to 0.15mm (+12.4 percent) wider than specified.
- The refractive index of the acrylic glass has not been considered in the planning stage.

While the first problem has been fixed with additional mountings, the second and the third problem have been solved by consecutive calibration steps. More details on this optimization process have been published by Eva Eggeling et al. [EHFU13].

Display Calibration

We opted for a calibration solution on-site because of influences of the environment, like lighting conditions and expansion due to heat development of all participating components. A major problem are inaccuracies in the direct measurement of display and parallax barrier parameters. It is not possible to make exact measurements of the distance from the display's pixel array to the parallax barrier without disassembling the displays.

In order to obtain initial values for the calibration, we need accurate measurements of a number of display and barrier parameters. Some parameters are scalar values (like the distance between two consecutive lines of a parallax barrier pattern), others are vectors (like the global position of the parallax barriers and the displays) that need to be registered to a global coordinate system. Performing direct measurements of vector valued parameters using a robotic arm was soon discarded due to the intricate construction of the mounting frame for the displays and the resulting inaccessibility of the corner points. Another idea to use the eye-tracker in combination with a laser distance measuring unit was discarded due to line of sight problems measuring display and barrier corner points.

Our calibration solution for this problem is an indirect system using a digital video camera. Since we already have initial values from direct measurements (where possible) and the digital construction of the driving simulator, we can optimize the parameters by evaluating a video feed from a camera. In this way, we do not optimize and calibrate the display parameters directly, but we calibrate the end result.

Since the calibration has to be performed for each display separately, it requires a custom mounting for the camera to be able to account for horizontal rotation around the optical center of the camera. The optical center of the camera was measured using a Panosaurus device from Gregwired. After manufacturing an appropriate mounting for the camera – the camera needs to be

rotated horizontally around it's center (no vertical rotation is necessary) – it was attached to the driving simulator. Since we already had a mounting for the calibration pattern of the eye-tracking system, only minor modifications were needed for the camera mount to be placed exactly in the position of left and right eye (separated by 65mm).

The parallax barrier display calibration consists of several steps to be performed consecutively: video mask determination, color calibration, and display calibration.

Video Mask Determination As the first step, we let the display system render two images and capture them with the video camera. One image is completely black; the other one is completely white, as shown in Figure 6.13 (left). We perform this step in order to create a mask, which is used in all subsequent steps to restrict the calibration to the display's area and to ignore irrelevant parts of the video images (e.g., interior parts, chassis parts, ...). For each pixel of the mask, a value between zero and one is stored to indicating whether the pixel shows a part of the display (1.0) or not (0.0 otherwise). The resulting binary mask is shown in Figure 6.13 (right).

Figure 6.13: We are only using clipped / weighted video input for the parallax barrier display calibration. The corresponding mask is generated in the first step. Although the automatically generated mask is sufficient, we advise to use a manually clipped mask as shown on the right hand side.

Color Calibration After video mask determination, the next step is to perform tests with a single color to compare input colors (sent to a display) with output colors (captured by the video camera) while taking the generated video mask into account. This step is performed for a configurable amount of colors (256 per default) spread evenly in the RGB color space. An example with 16 colors is shown in Figure 6.14.

This step is necessary in order to account for color deviations. Color deviations occur due to non-calibrated hardware (displays and camera) and due to color shifting effects caused by capturing images at an angle. The video mask allows us to directly compare the input color with every non-masked pixel from the captured images. Thus, the error function of the color calibration is the sum of per-pixel differences weighted with the normalized mask of the previous step. In our setting the colors blue and red show exceedingly few shifting effects (see Table 6.1).

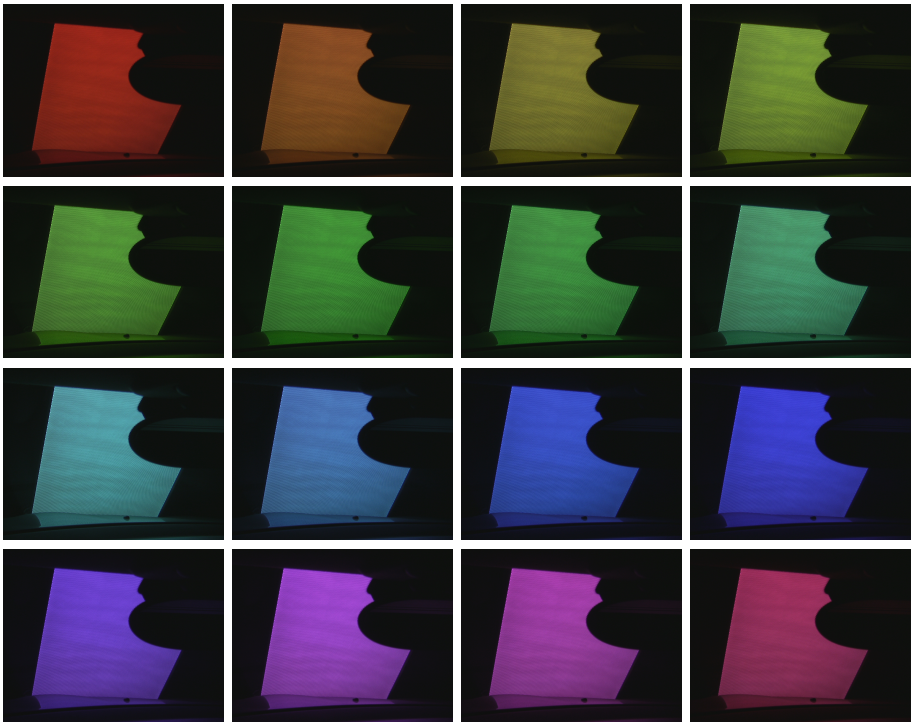


Figure 6.14: The color calibration step reduces color shifting effects caused by displays and camera. These 16 images show the captured results of fully saturated colors with varying hue spread evenly across the RGB color space.

Input color (RGB)	Output Color (RGB)	Error
(255,0,0)	(128,41,30)	143602.015625
(255,96,0)	(125,80,37)	148489.671875
(255,191,0)	(126,124,48)	171427.640625
(223,255,0)	(120,146,53)	178638.03125
(128,255,0)	(91,145,57)	157110.96875
(32,255,0)	(70,143,58)	161376.515625
(0,255,64)	(70,144,72)	163053.21875
(0,255,159)	(77,149,115)	171799.0625
(0,255,255)	(87,154,160)	194468.75
(0,159,255)	(77,120,169)	156308.0
(0,64,255)	(63,85,178)	130290.0
(32,0,255)	(65,68,182)	128080.859375
(128,0,255)	(103,70,180)	130436.625
(223,0,255)	(142,74,180)	154996.71875
(255,0,191)	(143,65,148)	153622.671875
(255,0,96)	(134,51,86)	142331.484375

Table 6.1: This table shows the error of the color calibration for the 16 images of Figure 6.14. The error function is the sum of per-pixel differences weighted with a normalized mask. The colors blue and red show the least shifting effects.

Display Calibration The next calibration steps are concerned with optimizing the barrier parameters. For this purpose, we start with sampling of the parameter space by using a sufficiently large interval around the theoretical parameters – the barrier parameters according to the construction plan. Finally, we use the best configuration of the previous step and perform a fine-tuning using a minimizing optimization routine.

The display calibration is performed using the following routine. An initial parameter vector consisting of all variables (barrier line width l_w, barrier line orientation l_α, barrier line vertical offset l_y, etc.) is created. A test image with the provided barrier parameters is calculated. It shows a black pixel, if it is visible from the eye position at which the camera is currently positioned. Otherwise a colored pixel (using the color determined in the color calibration step) is shown. The visibility is determined using an intersection test implemented in a shader program. It basically intersects a ray defined between a pixel center and the origin of the camera with the parallax barrier. In case the intersection point is covered by a barrier line, the pixel is not visible.

Having calculated the test image, the calibration tool displays the image and captures the video image to check the result. The goal is to have a completely black image. Therefore, the error function simply counts the number of non-black pixels weighted with the calibration mask.

This step is concerned with a sampling of the multidimensional parameter space to obtain uniformly distributed samples (see Figure 6.15, left). The final step uses the same error function and the best sample of the previous step to fine-tune the settings with a conjugate direction search routine. Figure 6.15 (right) shows the result of the final optimization.

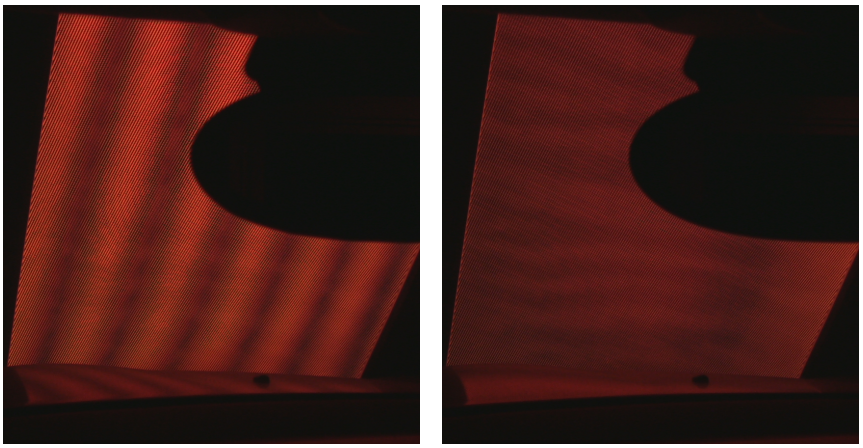


Figure 6.15: During display calibration, the multidimensional parameter space is optimized to obtain a good starting configuration for a search routine. On the left hand side the Figure shows a bad configuration captured during the sampling stage. The right hand side shows a good, final configuration.

In the course of several calibrations we made some interesting observations:

- The parameters *barrier line width* l_w and *barrier line distance* l_d are not independent. If the optimization routine does not have additional constraints, it returns a parameter setting, which corresponds to a black

test image. We solved this problem by introducing the linear constraint $2 \cdot l_w = l_d$, which we identified heuristically in a manual optimization test.

- The optimization routine returns the same configuration for both eye positions (with the numerical precision of the termination condition used by the optimization routine). Therefore, we have not been confronted with the problem of handling two different parameter settings and how to merge / combine them.
- The complete calibration runs completely automatically and can be started at any step, if previous results are available. In practice, the color calibration is performed once, for one eye position only. It is also possible to directly define a staring set of parallax barrier parameters to skip the time consuming sampling step in the second stage and directly start with the optimization routine.

Further Parameter Tuning First tests with real persons using the eye-tracking data revealed problems with ghosting effects, thus the pixel separation was not ideal. The initial idea to use an average color value for pixels classified as to be seen by both eyes proved to be part of the problem. After disabling these pixels, the ghosting effects almost disappeared, but not entirely.

Having ruled out any problems in the visualization pipeline, we encountered two problems.

1. The size of a pixel with respect to the line width of the barrier is rather large. Depending on the viewing angle, it is possible to have a pixel classified as visible by one eye, while nearly half of the pixel is visible for both eyes.
2. We also found the eye-tracking data to be the source of inaccuracy – the two camera system is struggling with partial occlusion and/or bad lighting conditions, resulting in errors in the magnitude of centimeters.

A solution to overcome these issues on the hardware-side would be to use more accurate eye-tracking, and displays with a larger resolution. However, even without upgrading the hardware, it is possible to further minimize the effect. The idea is to virtually increase the parameter value of the barrier line width. This way the virtual barrier becomes larger and more pixels are considered as non-separable. They are visible by both eyes or not visible at all. This solution reduces ghosting at the cost of resolution and brightness (as non-separable pixels are rendered black). The reduced resolution was observable in driving scenarios with detailed models and textures. Even though we did not encounter any problems concerning brightness (since the environment in the driving simulator is very dark), the situation can be improved by rendering non-separable pixel in a brighter gray.

Results & Runtime Generating and evaluating a single sample during the fully automatic display calibration takes about six seconds (including capturing an image with the camera and running the optimization). Depending on the parameter range, the whole process needs approximately $2\frac{1}{2}$ hours per display per eye (however, no user interaction is needed). The final barrier parameters

returned by the calibration routine have led to an error of 14–16 percent; i.e., approximately one-seventh of all pixels may show a wrong color for at least one eye (only in regions of non-uniform color).

The components to incorporate user tracking as well as autostereoscopic rendering have already been described. In detail, the perspective projection is directly handled by the InstantReality framework, while all parallax barrier calculations are performed using a multi-pass shader pipeline. In a first step, the images for left and right eye are rendered off-screen into two textures. A fragment shader in the following pass performs intersection tests (viewing ray $\leftrightarrow$ parallax barrier) at runtime from both eyes to all display pixels for the classification of the pixels. With the classification of the pixels at hand, the corresponding values from the textures are used to create the final image.

For the simulation part of the driving simulator, the visualization is a black box for visualizing a scene graph. All updates are sent using the External Authoring Interface (EAI) of the InstantReality framework. The visualization system on the other hand is not dependent on the simulation.

Evaluation

After finishing all calibration steps for the visualization system, the driving simulator has been used in a final user study with 21 test persons. Each test person was confronted with different, combined driving situations including inner city scenarios, overland tours and high-speed motorway driving. Next to a few specific questions after each driving maneuver, the participants had to answer a questionnaire afterwards. It consisted of simulator sickness questions, general questions about the overall system and specific questions on relevant subsystems.

Visualization System The visualization system has a large influence on the overall experience with the driving simulator. Parts of the user study have been conducted using 2D visualization, but since it is not easily possible to remove the parallax barriers, the participants had to deal with its visual influences. A large part of the scenarios was tested in 3D. The following results summarize the opinions of the participants.

On a scale from *unrealistic* to *realistic*, 16 out of 21 participants (about 76 percent) evaluated the visualization system as being *rather realistic* or *realistic*. Three participants deemed the visualization system to be *unrealistic*. Two of them had problems with ghosting effects and projection offsets, which are caused by problems of the eye-tracking system. The last participant of the three was not experiencing these effects, but suffered from general discomfort.

Using the same scale, 13 out of 21 participants (about 62 percent) experienced the representation of the environment (i.e., the modeled 3D scene) as *rather realistic* or *realistic*. One of the participants voted with *unrealistic*. This is likely due to problems with ghosting effects and projection offsets – the participant was one of the three experiencing the visualization system as unrealistic. From all 21 participants, 20 (about 95 percent) answered neutral or positive when asked to rate the driving simulator towards usefulness – 15 of them rated positive. The remaining participant was the one suffering from general discomfort.

A total of 14 participants (about 67 percent) would definitely partake in another study using this driving simulator – two participants would not. When asked to recommend the driving simulator to other drivers, 17 (about 81 percent) participants would do so. One participant answered negative when asked which impression of the driving simulator other drivers would get.

Answering open questions, one participant deemed the 3D visualization as being the most useful part of the driving simulator. On the other hand, participants had problems with dizziness, ghosting effects, the parallax barrier lines as well as stifling air. One participant struggling with ghosting effects had difficulties in perceiving speed with the 2D visualization, but had a better perception in 3D. We also had a suggestion to change the color of the parallax barrier lines from black to some color with less contrast. Regarding the choice of the preferred visualization, we had an almost neutral result between 2D and 3D.

Simulator Sickness Questionnaire Simulator sickness and motion sickness result in feelings of nausea, dizziness, vertigo, and sweating (among other symptoms). They are generally the result of the discrepancy between simulated visual motion and the sense of movement stemming from the vestibular system [BBI13]. In stationary simulators, the visual system receives information that suggests movement (e.g., roadway scenes passing by the viewer), yet the vestibular system interprets a stationary status. This discrepancy is the cause of simulator sickness in many people.

Robert S. Kennedy et al. proposed a Simulator Sickness Questionnaire (SSQ) that is widely used to asses sickness in simulators [KLBL93]. This questionnaire is derived from a motion sickness questionnaire and asks participants to score 16 symptoms (of three general categories: oculomotor, disorientation, and nausea) on a four point scale (0-3). Weights are assigned to each of the categories and summed together to obtain a single score.

The average values of the driving simulator are listed in Table 6.2. They are evaluated according to the handbook of Donald L. Fisher, Matthew Rizzo and Jeff K. Caird [FRC11]. The table includes comparative values published by Jukka Häkkinen et al. [HPTN06], who investigated simulator sickness in virtual display gaming in stereoscopic and non-stereoscopic situations.

	Parallax Barrier	Virtual 2D	Virtual 3D
nausea	34.9	11.8	29.9
oculomotor	49.8	14.0	26.9
disorientation	79.5	21.1	41.1

Table 6.2: The driving simulator has been evaluated using the Simulator Sickness Questionnaire [KLBL93]. In order to interpret the scores comparative values for non-stereoscopic displays (Virtual 2D) and stereoscopic displays using shutter techniques (Virtual 3D) are included [HPTN06].

6.3 Summary

This chapter focused on the final step in the shape processing pipeline discussed within this thesis: visualization techniques. In the first section, a system to seamlessly project onto non-planar surfaces was presented. It can be used to create dynamic, reconfigurable spaces influencing the behavior of groups and individuals. The subsequent section focused on an immersive, autostereoscopic visualization system for a driving simulator. Especially the aspect of comparability between real driving and driving in a simulator represented the rationale behind using an autostereoscopic system without the need to wear special glasses.

Since both systems heavily differ in their requirements, this chapter showed the diverse application fields for visualization.

Chapter 7

Conclusion & Future Work

This thesis presents work in the field of shape processing for content generation. It consists of shape descriptions, modeling paradigms, inverse modeling techniques, software engineering, computer graphics, and visualization technologies.

The foundations of this thesis are several scientific articles and conference contributions. Contributions have been made in the field of generative modeling, especially with the meta-modeler Euclides. A system to analyze digitized objects in terms of changes and damages as well as a novel shape modeling approach represent the contributions in the context of inverse modeling techniques. Two contributions dealing with projections onto non-planar surfaces and an autostereoscopic visualization system for a driving simulator conclude the chapter on visualization technologies.

Finally, this chapter also focuses on future work in the fields of generative modeling, inverse modeling techniques, and visualization technologies.

Contents

7.1 Generative Modeling

In the context of generative modeling, the contributions of this thesis are the meta-modeler approach Euclides, and a generative description of wedding rings together with a serverside rendering framework for the GML.

The general nature of the generative modeling paradigm to use formal construction rules to create highly complex shapes has a drawback. A generative description has to be re-implemented on every platform needed. This is problematic in larger systems, where different domain-specific languages and tools come into play (e.g., in the context of product mass customization). The effort to implement and maintain several generative descriptions is significant.

The meta-modeler approach Euclides helps to overcome this negative aspect of generative modeling and its explicit analogy of 3D modeling and programming [USF10]. The problem is solved by using a consistent intermediate representation serving as a basis for back-end exporters to different languages and different platforms. Due to its high-level representation of the input code, the level of abstraction can be preserved after translation to target code. Additionally, comments in the translated code (including statements of the input code) greatly increase its readability and allow for easy reuse and debugging. Amongst others, the compiler offers to output differentiated Java code to be used for inverse modeling tasks, and GML code. It is the first complete translator to a PostScript dialect, covering all control flow statements [SSUF10b].

Another aspect is the significant inhibition threshold that arises from the need to use a programming language. It introduces a new dimension of complexity and further dependencies. This problem is tackled by using JavaScript as a beginner-friendly, yet powerful language.

Based on Euclides' flexible libraries, complex examples, like an amphitheater or a cathedral, can be created with just a few hundred lines of code (see Figure 7.1).

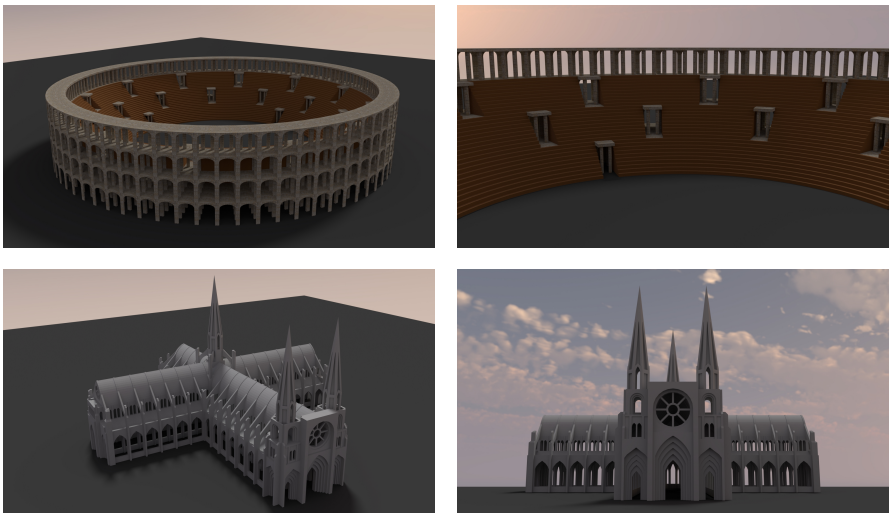


Figure 7.1: The photorealistic renderings of Euclides' amphitheater and cathedral examples are produced by the Cycles render engine of Blender.

The photorealistic renderings of amphitheater and cathedral are produced by the Cycles render engine of Blender.

Advantages of the presented approach can be observed in service-oriented environments. The potential execution of arbitrary code represents a significant security threat. In Euclides this threat can be reduced by security features of the Java virtual machine and by exercising control over the use of native code. Many security relevant aspects of programming require access to system resources. This functionality can be made available in Euclides via native code of the target platform. By restricting the use of native code to trusted libraries, security risks can be reduced.

Another contribution is the interactive interpreter based on the already existing compiler [SUF12]. Instead of creating large proportions of new code, whose behavior has to be consistent with the already existing compiler, an interpreter reusing most parts of the compiler's front- and back-end has been implemented.

The cHASER solution for the GML is a server-side rendering approach for applications tailored towards jewelers and wedding ring studios [SBEF14]. It allows the rendering to be shifted away from the client – it uses X3DOM technology to display a combination of simple proxy geometry and rendered images. This approach offers the advantage of preserving the interactivity of navigation, while at the same time ensuring rendering quality, which is independent of the client's hardware. By using rendered images instead of geometry, the intellectual property of the generative description is protected.

7.2 Inverse Modeling

This thesis presents two contributions in the area of inverse modeling. Analysis operations for digitized artifacts can identify possible changes and damages. By using this approach, new possibilities for shape modeling can be used to transfer features from one shape to another.

Using the approach to identify instances of a generative description in real-world data sets presented by Torsten Ullrich and Dieter Fellner, it is possible to automatically combine generative descriptions with reconstructed objects for nominal/actual value comparisons [SUSF11]. The reference surface is constructed using a generative description whose accuracy and systematics describe the semantic properties of an object, whereas the actual object is a laser scan or photogrammetric reconstruction without any additional semantic information.

The approach to analyze digitized objects allows to modify existing 3D shapes. While high-level descriptions can be used to resemble real-world objects, they can also be used to create new ones [SUF14] by transferring features from one shape to another. Shapes using both low-level details and high-level shape parameters can be designed in this way.

7.3 Visualization Technologies

In the field of visualization technologies, the contributions of this thesis consist of a method for tiled projections onto non-planar surfaces, and parallax barrier displays for autostereoscopic visualizations.

The work on non-planar projections is concerned with an efficient method to project a coherent, seamless and perspectively corrected image from one particular viewpoint using an arbitrary number of projectors [KSHF13]. Determining and compensating the geometric relationship between the overlapping projectors is the key component of the framework. Using a simple checkerboard pattern, the geometric relationship between the projectors is measured. It can easily filter out minor irregularities of the screen surface, such as detailed winkles. The detected corner point coordinates of the checkerboard pattern are passed to a Processing script, and then used to create a uniform display area. The projection display system supports a quick calibration adapting to non-standard configurations dealing with arbitrary projection environments.

The parallax barrier approach describes the construction, design, optimization, calibration and evaluation of a parallax barrier visualization system for a driving simulator [SPH+16]. Two main components of the system are concerned with calibration of the parallax barrier and the runtime environment (the actual visualization). After having optimized the pattern for each display, calibration has to be performed due to deviations of the manufactured acrylic glasses (see Figure 7.2) from the ideal ones.

The visualization including perspective projection and parallax barrier calculations of a scene are handled using the InstantReality framework. The main advantage of this approach lies in its autostereoscopic nature – no special glasses are needed for the experience.

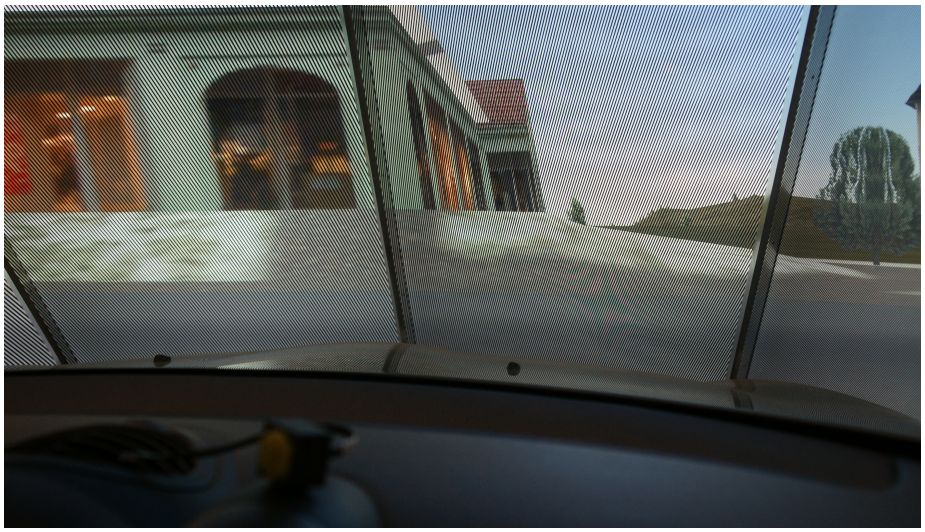


Figure 7.2: The individual parallax barriers for each display are manufactured using acrylic glass. Mounted in precise distance in front of the displays, they are used by the system to separate pixels for left and right eye.

7.4 Future Work

Concrete ideas to continue the work in the fields of generative modeling, inverse modeling techniques, and visualization technologies are manifold. The following

sections give a brief insight into what directions further work can take (or is already taking).

7.4.1 Generative Modeling

In the context of the Euclides framework, further work can be carried out to cover more target platforms. Especially CAD software used in the design process of products can benefit from automation in the form of generative modeling. Additionally, future work should focus on performance of the created code. Some of Euclides' data types could be mapped to native data types of the target languages. This may be implemented as a "performance mode" in addition to a "debugging mode". More high-level libraries for modeling operations, like the creation of subdivision surfaces or CSG operations, would simplify the workflow of creating expressive generative models and should be among the next development steps. Another idea is to offer Euclides' functionality directly in the Web to create a generative modeling as a service infrastructure.

The cHASER server-side rendering approach for the GML is limited to the included OpenGL rendering pipeline. Ongoing work on using the Cycles render engine of Blender to create photorealistic renderings already produced some results (see Figure 7.3). A Blender exporter on the GML side exports all necessary data (geometry, materials, displacements, ...) as a Python script. This script is used by a custom library for Blender to import all data and creating the renderings. It is planned to automatically pre-create renderings of a certain parameter space to be used for a web-based ring configurator.

Figure 7.3: The photorealistic renderings of the JohannKaiser ring design space are produced by the Cycles render engine of Blender. A Blender exporter on the GML side exports all necessary data (geometry, materials, displacements, ...) as a Python script. On the Blender side, a custom library is capable of importing all data and creating the renderings.

7.4.2 Inverse Modeling

Future work regarding the nominal/actual value comparison of digitized objects should focus on two aspects. On the one hand, the deficiencies in the output mesh caused by displacing vertices in normal direction should be addressed. Depending on the curvature of the surface, this could be fixed by taking the adjacent primitives into account and displacing vertices in averaged normal direction – or even by using pre-computed (arbitrary) directions. On the other

hand, the whole process of exporting geometry to X3DOM could be made obsolete by implementing an X3DOM exporter for Euclides. In this way, the generative description could be evaluated directly in X3DOM, greatly reducing the file size at the cost of more computational effort. Depending on the resolution of the displacements, this approach could also be used for mesh compression, or progressive mesh streaming purposes. The generative description serves as a base mesh, while the details (stored in a displacement map) can be progressively streamed to the client.

7.4.3 Visualization Technologies

The presented framework to seamlessly project onto bent surfaces can only deal with limited (pre-defined) changes of the projection surface. Methods for recognizing the surface geometry and adapting the projection based on real-time measurements of the quad mesh texture would further enhance the versatility of the approach. While interrupting a visual installation with checkerboard patterns is not feasible, a possibility would be to display the pattern only for a very short time – imperceptible for the human visual system. With a synchronized camera and an optimized runtime of the pattern detection algorithm, a real-time adjustment to the deformed surface is possible.

Ongoing work to further develop the versatility of the driving simulator includes the implementation of virtual side mirrors (see Figure 7.4), as well as a rear-view mirror. The side mirrors are implemented using rendered textures embedded into side mirror geometry on the side displays. In contrast to virtually implementing the side mirrors, the rear-view mirror is the original mirror used in conjunction with an additional display mounted in front of the rear window inside the car. The side-mirrors react to movements of the driver, but it's geometry is static (it cannot be adjusted like in a real car).

Figure 7.4: The driving simulator is in the process of being equipped with virtual side mirrors. They are implemented using rendered textures embedded into side mirror geometry on the side displays.
(Source: Philipp Quinz, 2017)

A question of future research is what to do with pixels that are visible by both eyes. First tests using blended pixel values of both images for these pixels resulted in significant ghosting effects. Since these effects are less noticeable in homogeneous regions, an idea would be to use the pixels only for these areas to obtain more overall brightness. Another option would be to use the gaze

direction provided by the eye-tracking system to restrict the parallax barrier calculation to a certain region. A two-dimensional rendering could be used for peripheral areas, while performing intricate barrier calculations only in an area around the gaze direction. While this kind of rendering may reduce the simulator sickness, it would require a very accurate gaze direction at very high update rates. In order to create more realistic driving scenarios, dazzling effects play an important role. These effects are difficult to implement using standard display hardware because of the low dynamic range of the light source. The creation, or approximation of such High Dynamic Range (HDR) effects could be further investigated.

Appendices

Appendix A

Euclides Language Elements

Since the language Euclides is closely related to JavaScript, its source consists of a sequence of statements consisting of expressions and comments.

Euclides supports short and long comments, similar to other languages. Statements in Euclides are all constructs consisting of a line (or several lines) of code. Where expressions produce a value and can be written wherever a value is expected, a statement performs an action. Wherever Euclides expects a statement, also expressions can be used. Such a statement is called an expression statement.

This chapter gives an overview of all comments, statements, and expressions available in Euclides.

Contents

A.1 Comments

The language Euclides has two different kinds of comments, which can be at arbitrarily positioned within the source code: short comments and long comments. A short comment starts with `//` and ends at the end of the line in which it started. A long comment starts with `/**`, can include several line breaks and ends with `*/`. In contrast to other languages that support a similar functionality (such as C, C++, Java, etc.), Euclides does not support comments starting with `/*` instead of `/**` (see Listing A.1).

```
1  /**
2   * This program is free software; you can redistribute it and/or
3   * modify it under the terms of the GNU General Public License
4   * as published by the Free Software Foundation; version 2 of
5   * the License.
6   *
7   * This program is distributed in the hope that it will be
8   * useful, but WITHOUT ANY WARRANTY; without even the implied
9   * warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
10  * PURPOSE. See the GNU General Public License for more details.
11  */
12
13 var x = 1.0;      // Each of these comments ends at the end of
14 var y = x + 2.0; // the line in which it has started.
```

Listing A.1: A long comment starts with `/**` and ends with `*/`. It can include several line breaks. Short comments in Euclides start with `//` and end at the end of the line in which they started.

A.2 Statements

In Euclides, statements are everything that makes up a line (or several lines) of code. The following statements are available in Euclides:

- empty statement
- block statement
- function declaration statement
- variable declaration statement
- expression statement
- if statement
- for statement
- for-in statement
- while statement
- do-while statement
- switch statement
- continue statement

- break statement
- return statement
- throw statement
- try statement
- annotation statement
- native code statement

Where an expression produces a value and can be written wherever a value is expected, a statement performs an action.

A.2.1 Empty Statement

The empty statement is the simplest statement – it has no semantic meaning and can be omitted. A rail-road-diagram of the statement is shown in Figure A.1.

Figure A.1: The simplest Euclides statement is the empty statement. It consists of a single semicolon.

A.2.2 Block Statement

The block statement combines an arbitrary number of statements (see Figure A.2.

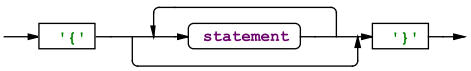


Figure A.2: A block statement is used to place several statements at a place, where only one statement is allowed to be placed.

The order of the statements within a statement block defines the order of execution. Concerning the visibility of variables Euclides has the same scoping rules as JavaScript; i.e., in contrast to other languages that support a similar functionality of statement blocks (such as C, C++, Java, etc.), Euclides does not start a new scope within a statement block; i.e., any identifier defined within a statement block is semantically equivalent to an identifier defined outside. An example of the statement block is shown in Listing A.2.

```
1 var x, y;
2 if (true) {
3     x = 1;
4     y = -1;
5 } else {
6     x = -1;
7     y = 1;
```

```
8 }
```

Listing A.2: A block statement is delimited by curly brackets and is used to place several statements at a place, where only one statement is allowed to be placed. In contrast to other languages it does not start a new scope.

A.2.3 Function Declaration Statement

The function declaration statement defines a new, non-anonymous function (see Figure A.3).

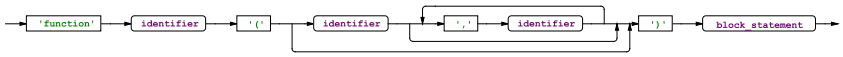


Figure A.3: The function declaration statement defines a new function and assigns it to a newly declared variable. The function / variable name has to be a free identifier. A function may have an arbitrary number of parameters.

The function name has to be a valid identifier. A function declaration statement can be replaced by a variable declaration statement and an assignment statement that assigns an anonymous function to the newly declared variable (see Listing A.3). In Euclides it is good practice to declare the functions of a library globally using function declaration statements and to declare non-library functions using variable declaration statements and anonymous functions.

```
1  function sqr(x) {
2      var y = x * x;
3      return y;
4  }
5
6  function cub(x) {
7      var y = x * x * x;
8      return y;
9  }
10
11 var pot = function(x,y) {
12     var i, z=1;
13     for (i = 1; i <= y; i++)
14         z*=x;
15     return z;
16 }
17
18 var a, b, c, d;
19 a = sqr(4);
20 b = cub(6);
21 c = pot(a,b);
22 d = pot(cub(3.7), 4);
```

Listing A.3: A function declaration statement defines a new, non-anonymous function. It can be replaced by a variable declaration statement and an assignment statement that assigns an anonymous function to the newly declared variable.

A.2.4 Variable Declaration Statement

The variable declaration statement declares a new variable. The default value of every, newly declared variable is `null` (see Figure A.4).

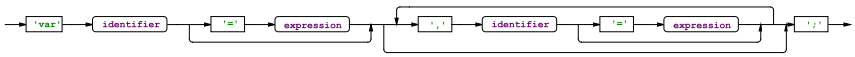


Figure A.4: The variable declaration statement declares a new variable. Furthermore, it offers the possibility to define its value using an expression.

The variable name has to be a valid identifier. Variable assignments can be performed in the same statement, or at a later point (see Listing A.4).

```
1 var a = 1, b = 2, c = 4, d = 8;
2 var answer;
3 a = 12;
4 b = 52;
5 c = 41;
6 d = a + b * c;
7 answer = "Result: " + d;
```

Listing A.4: A variable declaration statement declares a new variable. Variable assignments can be performed in the same statement, or at a later point.

A.2.5 Expression Statement

The expression statement serves as a wrapper statement to execute an expression (see Figure A.5).

Figure A.5: The expression statement executes an expression. If the expression is missing, the statement becomes an empty statement.

The execution order of an expression statement is defined by the operator precedence. Listing A.5 shows example expressions.

```
1 var x, y;
2 x = 1;
3 y = 2*x*x + 4*x + 5;
```

Listing A.5: An expression statement serves as a wrapper statement to execute an expression. The execution order of an expression statement is defined by the operator precedence.

A.2.6 If Statement

The if-statement controls conditional branching (see Figure A.6).
The conditional expression is expected to return a Boolean value. Any non-Boolean value will be converted to a Boolean value according to the following rules:

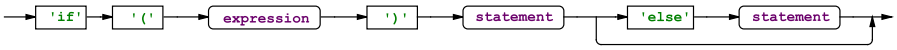


Figure A.6: The if-statement is one of the control flow statements supported within Euclides.

type	conversion
array `a`	`false`
number `n`	`n != 0.0`
object `o`	`true`
string `s`	`(s != null) && (s.length > 0)`

Listing A.6 shows a simple if-statement with an else clause.

```
1 var x, y;
2 if (true) {
3     x = 1;
4     y = 2;
5 } else {
6     x = "true";
7     y = false;
8 }
```

Listing A.6: The if-statement controls conditional branching. Any non-Boolean value will be converted to a Boolean value.

A.2.7 For Statement

The for-statement consists of a mandatory assignment expression, an optional condition expression, an optional loop expression and a loop body (see Figure A.7). The loop body is a single statement.

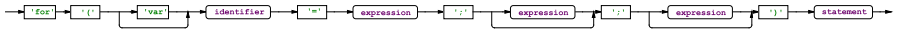


Figure A.7: The for-statement is one of the control flow statements supported within Euclides. It can be used to construct loops that must execute a specified number of times.

Prior to any other element of the for-statement, the assignment expression is executed only once. Control then passes to the condition expression. The condition expression is executed prior to each iteration of statement, including the first iteration. The loop body statement is executed, only if the condition expression evaluates to true. Any non-Boolean will be converted to a Boolean value according to the following rules:

type	conversion
array `a`	`false`
number `n`	`n != 0.0`
object `o`	`true`
string `s`	`(s != null) && (s.length > 0)`

At the end of each iteration of the loop body statement the loop expression is executed. A `for` loop terminates when one of these statements is executed

within the loop body: break statement, thrown statement, return statement. A continue statement in a `for` loop terminates only the current iteration. A simple for-statement is shown in Listing A.7.

```
1  var sum = 0;
2  for(var i = 1; i<=10; i++) {
3      sum += i;
4  }
```

Listing A.7: The for-statement can be used to construct loops that must execute a specified number of times.

A.2.8 For-In Statement

The for-in statement executes the loop body statement repeatedly and sequentially for each element in the range expression (see Figure A.8).

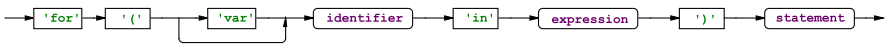


Figure A.8: The for-in statement can be used to construct loops that must iterate through a range defined by an array.

A `for` loop terminates when one of these statements is executed within the loop body: break statement, throw statement, return statement. A continue statement in a `for` loop terminates only the current iteration. A simple for-in statement is shown in Listing A.8.

```
1  var index;
2  var sum = 0;
3  for(index in [1, 2, 3, 4, 5, 6, 7]) {
4      sum += index;
5  }
```

Listing A.8: The for-in statement executes the loop body statement repeatedly and sequentially for each element in the range expression.

A.2.9 While Statement

The while statement executes a statement repeatedly until the condition expression evaluates to `false` (see Figure A.9).

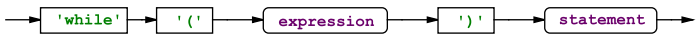


Figure A.9: The while statement is a control flow statement, which belongs to the class of loop statements.

The test of the condition expression takes place before each execution of the loop – a while loop executes zero or more times. It terminates when one of these statements is executed within the loop body: break statement, throw statement, return statement. A continue statement in a while loop terminates only the current iteration. Simple examples of while loops are shown in Listing A.9.

```
while (false) {
    // dead code
}

var i = 1;
while (i < 100) {
   i = i*i;
}

while (true) {
    // infinite loop
}
```

Listing A.9: The while statement executes a statement repeatedly until the condition expression evaluates to `false`.

A.2.10 Do-While Statement

The do-while statement executes a statement repeatedly until the condition expression evaluates to `false` (see Figure A.10).

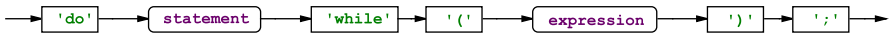


Figure A.10: The do-while statement is a control flow statement, which belongs to the class of loop statements.

The do-while statement exhibits properties similar to a while statement – a simple example is shown in Listing A.10.

```
var i = 1;
do {
    i = i*i;
} while (i < 100);
```

Listing A.10: The do-while statement executes a statement repeatedly until the condition expression evaluates to `false`.

A.2.11 Switch Statement

The switch statement allows selection among multiple subsections of code, depending on the value of an expression. It consists of a selection expression, zero or more case blocks and an optional default block (see Figure A.11).
If a matching expression is found in a case block, control is not impeded by subsequent case or default labels. The break statement is used to stop execution and transfer control to the statement after the switch statement. Without a break statement, every statement from the matched case label to the end of the switch, including the default, is executed. An example of a switch statement is shown in Listing A.11.

```
var small_numbers = 0;
var big_numbers = 0;
var zeros = 0;

for(var i in [0, 1, 2, 3, 4, 5, 6])
{
```

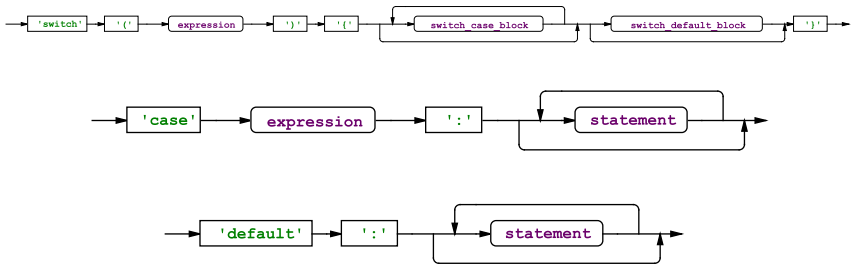


Figure A.11: A switch statement (top) consists of a selection expression, zero or more case blocks and an optional default block. The case block (middle) starts with the keyword `case` followed by an expression, which is compared to the selection expression, and optional statements, which are executed, if the comparison reveals equality. The optional default block (bottom) at the end of a switch statement is executed, if no case block has been executed before.

```
 7       switch(i)
 8       {
 9           case 0:
10               zeros++;
11           case 1:
12           case 2:
13           case 3:
14               small_numbers++;
15               break;
16           case 4:
17           case 5:
18           case 6:
19               break;
20           default:
21               big_numbers++;
22       }
23   }
```

Listing A.11: The switch statement allows selection among multiple subsections of code, depending on the value of an expression.

A.2.12 Continue Statement

The continue statement forces transfer of control to the controlling expression of the smallest enclosing do-while, for, for-in, or while loop. A rail-road-diagram of the statement is shown in Figure A.12.

Figure A.12: The continue statement belongs to the class of jump statements.

The continue statement can only be used within a loop (for, for-in, do-while, while) or in a switch-statement. Any remaining statements in the current iteration of the smallest enclosing loop are not executed. An example showing the use of the continue statement within a while loop is shown in Listing A.12.

```
1  var i = 0;
2  while (i < 100) {
3      i++;
4      if (i < 50)
5          continue;
6  }
```

Listing A.12: The continue statement forces transfer of control to the controlling expression of the smallest enclosing do-while, for, for-in, or while loop.

A.2.13 Break Statement

The break statement ends execution of the nearest enclosing loop or case block in which it appears. Control passes to the statement that follows the end of the statement, if any. A rail-road-diagram of the statement is shown in Figure A.13.

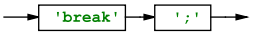


Figure A.13: The break statement belongs to the class of jump statements.

The break statement can only be used within a loop (for, for-in, do-while, while) or in a switch-statement. Within the switch statement a break statement is used to stop execution and transfer control to the statement after the switch statement. Without a break statement, every statement from the matched case label to the end of the switch statement, including the default block, is executed. In loops, the break statement ends execution of the nearest enclosing for, for-in, do-while, while statement. Control passes to the statement that follows the ended statement, if any. An example showing the use of the break statement within a for loop is shown in Listing A.13.

```
1  for (var i = 1; i < 10; i++) {
2      if (i == 4)
3          break;
4  }
```

Listing A.13: The break statement ends execution of the nearest enclosing loop or case block in which it appears.

A.2.14 Return Statement

The return statement terminates the execution of a function and returns control to the calling function. Execution resumes in the calling function at the point immediately following the call. A rail-road-diagram of the statement is shown in Figure A.14.

Figure A.14: The value of the expression is returned to the calling function. If the expression is omitted, the return value of the function is undefined.

A function can have any number of return statements. Listing A.14 shows the use of the return statement in different functions.

```
function sqr(x) {
    var y = x * x;
    return y;
}

function cub(x) {
    var y = x * x * x;
    return y;
}

var pot = function(x,y) {
    var i, z=1;
    for (i = 1; i <= y; i++)
        z*=x;
    return z;
}

var a, b, c, d;
a = sqr(4);
b = cub(6);
c = pot(a,b);
d = pot(cub(3.7), 4);
```

Listing A.14: The return statement terminates the execution of a function (anonymous and non-anonymous) and returns control to the calling function.

A.2.15 Throw Statement

Throw and try statement implement exception handling in Euclides. A railroad-diagram of the throw statement is shown in Figure A.15.

Figure A.15: The expression of a throw statement captures data for a catch block of an exception handling mechanism.

A throw statement signals that an exceptional condition – often, an error – has occurred in a try block. An example showing the use of the throw statement within a try block is shown in Listing A.15.

```
try {
    // some code

    throw "an error appeared";

    // the code after the throw statement is not executed

} catch (error) {

    // now error contains the string "an error appeared";

}
```

Listing A.15: A throw statement signals that an exceptional condition has occurred in a try block.

A.2.16 Try Statement

Try and throw statement implement exception handling in Euclides. A rail-road-diagram of the try statement is shown in Figure A.16.

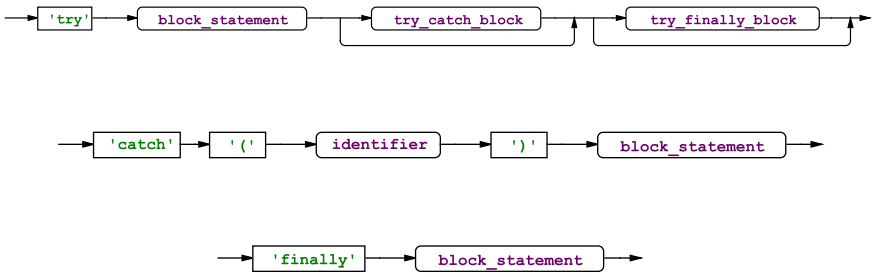


Figure A.16: The statement block contains the "critical" code that may throw an exception. Code after the throw statement will not be executed. The catch block catches the thrown exception and assigns the thrown expression– often data about an error, an error code, etc – to the identifier. The finally block is always executed.

A throw statement signals that an exceptional condition – often, an error – has occurred in a try block. An example showing the use of the try statement is shown in Listing A.15.

A.2.17 Annotation Statement

The annotation statement in Euclides is a standardized way to define properties. A rail-road-diagram of the annotation statement is shown in Figure A.17.

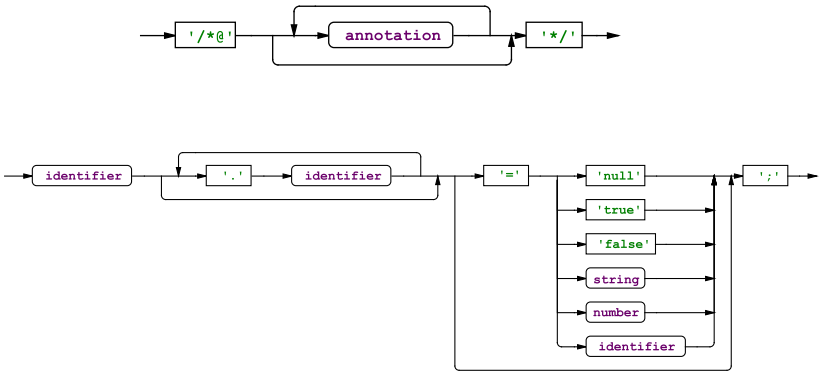


Figure A.17: The annotation statement is a formal comment; i.e., it is parsed and has to conform to the grammar rules (therefore it is formal), but it is not interpreted (as it is a comment). Each annotation within an annotation statement looks like a key-value-pair.

Annotations are used by the compilation pipeline (in future releases). Currently, no annotations are defined.

A.2.18 Native Code Statement

The native code statement can be used to include platform specific code. This feature is necessary for including specific functionality of the platform. A railroad-diagram of the throw statement is shown in Figure A.18.

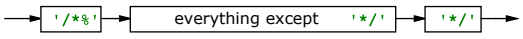


Figure A.18: A native code statement looks like a normal JavaScript multi-line comment. In Euclides it is a black box of code inlined into the compilation.

The Euclides compiler offers several compilation targets. The inclusion of native code statements may compromise cross platform development. This feature is intended to be used by designers of new libraries and compilation targets. An example showing the use of the native code statement to output text is shown in Listing A.16.

```
1 function print(msg) {
2     /*% System.out.println(usr_msg.toString()); */
3 }
```

Listing A.16: The native-code statement is necessary for including platform specific functionality like text output.

A.3 Expressions

Wherever Euclides expects a statement, an expression can also be used. Such a statement is called an expression statement. The reverse does not hold: a statement cannot be written where Euclides expects an expression. For example, a for statement cannot become the argument of a function. This chapter describes all Expressions supported by Euclides:

- Operators: Euclides, like JavaScript, has unary and binary operators, as well as a single tertiary operator.
- Identifiers: Identifiers are used in statements and expressions to name variables and functions.
- This: The This reference plays an important rule in Euclides.
- Constants: Constants are literals like `true` and `false`.
- Arrays: Euclides fully supports arrays.
- Objects: The concept of objects is supported in Euclides.
- Functions: Functions itself are also expressions.

A.3.1 Unary Operators

Unary operators can be divided in prefix (preceding an unary expression) and postfix (after an unary expression) operators. Unary prefix operators are:

- delete (`delete`)
- void (`void`)
- typeof (`typeof`)
- new (`new`)
- increment (`++`)
- decrement (`--`)
- add (`+`)
- subtract (`-`)
- bitwise not (`~`)
- not (`!`)

Unary postfix operators are:

- increment (`++`)
- decrement (`--`)

A.3.2 Binary Operators

Binary operators can be divided in arithmetical, assignment, equational, logical, multiplicative, relational, and shift operators with a specific precedence (and even inner precedence). The following lists of binary operators starts with the group of lowest precedence. Assignment operators are:

- assign (`=`)
- assign multiply (`*=`)
- assign divide (`/=`)
- assign modulo (`%=`)
- assign add (`+=`)
- assign subtract (`-=`)
- assign bitwise left shift (`<<=`)
- assign bitwise right shift (`>>=`)
- assign zero fill bitwise right shift (`>>>=`)
- assign bitwise and (`&=`)
- assign bitwise xor (`^=`)
- assign bitwise or (`!=`)

Logical operators are:

- or (`||`)
- and (`&&`)
- bitwise and (`&`)
- bitwise or (`|`)
- bitwise xor (`^`)

Equational operators are:

- equal (`==`)
- not equal (`!=`)
- strict equal (`===`)
- not strict equal (`!==`)

Relational operators are:

- less (`<`)
- greater (`>`)
- less equal (`<=`)
- greater equal (`>=`)
- instance of (`instanceof`)
- in (`in`)

Shift operators are:

- bitwise left shift (`<<`)
- bitwise right shift (`>>`)
- zero fill bitwise right shift (`>>>`)

Arithmetical operators are:

- add (`+`)
- subtract (`-`)

Multiplicative operators are:

- multiply (`*`)
- divide (`/`)
- modulo (`%`)

Please note that logical operators are not equal in precedence.

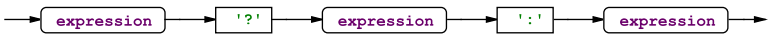


Figure A.19: The conditional operator is the only operator in Euclides that takes three operands. This operator is frequently used as a short-cut for the if statement.

A.3.3 Tertiary Operators

There is one tertiary operator available in Euclides – the conditional operator. A rail-road-diagram of the conditional operator is shown in Figure A.19.

A.3.4 Identifiers

In Euclides, the first character of an identifier must be a letter (lower, or upper case). Subsequent characters may be letters (again lower, or upper case), digits, or underscores (see Figure A.20).

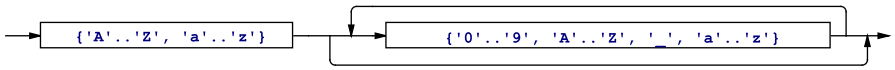


Figure A.20: Euclides allows the following characters in identifiers. Please note the limitations for the first character.

A.3.5 This Reference

The `this` reference plays an important role in Euclides - it holds a reference to the current execution context. This is important to keep in mind when using objects and functions (see Listing A.17).

```
1  var x = 2;
2
3  function sqr() {
4      var y = this.x * this.x;
5      return y;
6  }
7
8  var o = {
9      x : 3,
10     sqr : function() {
11         var y = this.x * this.x;
12         return y;
13     }
14 };
15
16 var a, b;
17 a = sqr(); // a is 4
18 b = o.sqr(); // b is 9
```

Listing A.17: The `this` reference holds the current execution context – it is used with functions and objects.

Since there is no outer environment context when using `this` globally, it holds the global environment.

A.3.6 Constants

Constant expression are `null`, `true`, `false`, as well as strings and numbers. Railroad-diagrams of strings (top) and numbers (bottom) are shown in Figure A.21.

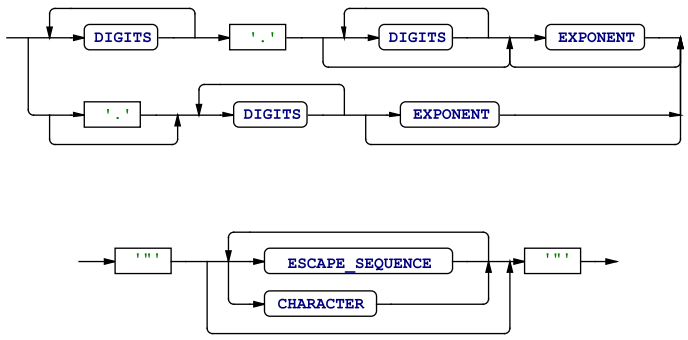


Figure A.21: Strings are enclosed with double quotes and can be of any character or escape sequence (top). Numbers may consist of digits, a decimal dot, and an exponent (bottom).

A.3.7 Arrays

In Euclides arrays are high-level, list-like objects and are used to store multiple values in a single variable (see Listing A.18).

```
1  var names = [
2      "Magdalena",
3      "Julia",
4      "Theresa",
5      42
6  ];
7
8  var favourite = names[0];
9
10 names[names.length-1] = "Letizia";
```

Listing A.18: Arrays in Euclides can hold different data types and they support the `length` property.

Some important properties of arrays are:

- Arrays can be created using the `new` keyword followed by a constructor.
- It is possible to have different data types in one array.
- Arrays support the `length` property.

A.3.8 Objects

In Euclides an object is a collection of properties (an association between a key and a value) (see Listing A.19).

```
1  var car = {
2    type : "Fiat",
3    model : 500,
4    color : "white"
5    getPrice : function() {
6      // do something to calculate the price
7      return this.model * 2;
8    }
9  };
10
11 var price = car.getPrice();
```

Listing A.19: Objects in Euclides are a collection of properties (key-value pairs).

Some important properties of objects are:

- Objects can hold any data types, also objects and functions.
- The `this` keyword in a function defined within an object refers to the environment of the object.

A.3.9 Functions

Functions can be defined anonymously, e.g., within an object. The syntax differs a bit from a function defined as a statement, as no identifier is used after the `function` keyword. A rail-road-diagram of the conditional operator is shown in Figure A.22.

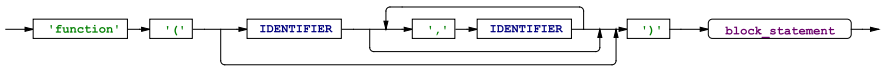


Figure A.22: Functions are defined anonymously (as an expression) without an identifier.

An example of an anonymous function is shown in Listing A.19.

Bibliography

[AA14] Marios C. Angelides and Harry Agius, editors. *Handbook of Digital Games*. John Wiley & Sons, 2014.

[AGFF09] M. Attene, D. Giorgi, M. Ferri, and B. Falcidieno. On converting sets of tetrahedra to combinatorial and pl manifolds. *Computer Aided Geometric Design*, 26:850–864, 2009.

[AGL05] James Ahrens, Berk Geveci, and Charles Law. Paraview: An End-User Tool for Large-Data Visualization. *Visualization Handbook*, 1:717–731, 2005.

[AL09] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on GPUs. *Proceedings of the Conference on High Performance Graphics*, 3:145–149, 2009.

[AS93] Günter Aumann and Klaus Spitzmüller. *Computerorientierte Geometrie*. BI-Wissenschafts-Verlag, 1993.

[ATAPvL11] Phillipa Avery, Julian Togelius, Elvis Alistar, and Robert Pieter van Leeuwen. Computational Intelligence and Tower Defence Games. *Proceedings of IEEE Congress on Evolutionary Computation (CEC)*, 13:1084–1091, 2011.

[Bab17] Norayr Babikian. 3D Space Pro - Become a better jeweler. online: http://www.3dspaceproinc.com, 2017.

[BBI13] Stacy A. Balk, Anne Bertola, and Vaughan W. Inman. Simulator Sickness Questionnaire: Twenty Years Later. *Proceedings of the International Driving Symposium on Human Factors in Driver Assessment, Training, and Vehicle Design*, 7:257–263, 2013.

[Beh17] Johannes Behr. Instant3dhub. online: http://instant3dhub.org/, 2017.

[BEJZ09] Johannes Behr, Peter Eschler, Yvonne Jung, and Michael Zöllner. X3DOM: A DOM-based HTML5/X3D Integration Model. *Proceedings of the 14th International Conference on 3D Web Technology*, 14:127–135, 2009.

[BFH05] René Berndt, Dieter W. Fellner, and Sven Havemann. Generative 3D Models: a Key to More Information within less Bandwidth at Higher Quality. *Proceedings of the 10th International Conference on 3D Web Technology*, 1:111–121, 2005.

[BHMT13] Alexandre Boulch, Simon Houllier, Renaud Marlet, and Olivier Tournaire. Semantizing Complex 3D Scenes using Constrained Attribute Grammars. *Proceedings of Eurographics Symposium on Geometry Processing*, 32:33–42, 2013.

[BK10] Mario Botsch and Leif Kobbelt. *Polygon Mesh Processing*. A K Peters, 2010.

[BSBK02] Mario Botsch, Stefan Steinberg, Stefan Bischoff, and Leif Kobbelt. Openmesh – a generic and efficient polygon mesh data structure. *Proceedings of OpenSG Symposium*, 1:1–5, 2002.

[BSK67] Theodore R. Bashkow, Azra Sasson, and Arnold Kronfeld. System Design of a FORTRAN Machine. *IEEE Transactions on Electronic Computers*, 16:485–499, 1967.

[BSK+12] René Berndt, Christoph Schinko, Ulrich Krispel, Volker Settgast, Sven Havemann, Eva Eggeling, and Dieter W. Fellner. Ring's Anatomy – Parametric Design of Wedding Rings. *Proceedings of the 4th International Conference on Creative Content Technologies*, 4:72–78, 2012.

[BTS+14] Matthew Berger, Andrea Tagliasacchi, Lee M. Seversky, Pierre Alliez, Joshua A. Levine, Andrei Sharf, and Claudio Silva. State of the Art in Surface Reconstruction from Point Clouds. *Eurographics 2014 - State of the Art Reports*, 1:161–185, 2014.

[Bun17] Bundesministerium für Bildung und Forschung (BMBF). Digitale Wirtschaft und Gesellschaft: Zukunftsprojekt Industrie 4.0. online: http://www.bmbf.de/de/zukunftsprojekt-industrie-4-0-848.html, 2017.

[BWS+07] Philip Benzie, John Watson, Phil Surman, Ismo Rakkolainen, Klaus Hopf, Hakan Urey, Ventseslav Sainov, and Christoph von Kopylow. A Survey of 3DTV Displays: Techniques and Technologies. *IEEE Transactions on Circuits and Systems for Video Technology*, 17:1647–1658, 2007.

[CC78] Edwin Catmull and Jim Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10:350–355, 1978.

[CGM+06] Andy Cedilnik, Berk Geveci, Kenneth Moreland, James P. Ahrens, and Jean M. Favre. Remote large data visualization in the paraview framework. *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization*, 6:163–170, 2006.

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956.

[CM06] Kate Compton and Michael Mateas. Procedural Level Design for Platform Games. *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference*, 2:109–111, 2006.

[CNGF04] Daniel Cotting, Martin Naef, Markus Gross, and Henry Fuchs. Embedding Imperceptible Patterns into Projected Images for Simultaneous Acquisition and Display. *Proceedings of the 3rd IEEE/ACM International Symposium on Mixed and Augmented Reality*, 3:100–109, 2004.

[CSLR01] Thomas H. Cormen, Clifford Stein, Charles E. Leiserson, and Robert L. Rivest. *Introduction to Algorithms*. B&T, 2001.

[CSS+11] Amaresh Chakrabarti, Kristina Shea, Robert Stone, Jonathan Cagan, Matthew Campbell, Noe Vargas-Hernandez, and Kirtsin L. Wood. Computer-Based Design Synthesis Research: An Overview. *Journal of Computing and Information Science in Engineering*, 11(2):1–10, 2011.

[CZGF05] Daniel Cotting, Remo Ziegler, Markus Gross, and Henry Fuchs. Adaptive Instant Displays: Continuously Calibrated Projections Using Per-Pixel Light Control. *Computer Graphics Forum*, 24:705–714, 2005.

[DS78] Daniel Doo and Malcolm Sabin. Behavior of Recursive Division Surfaces near Extraordinary Points. *Computer Aided Design*, 10(6):356–360, 1978.

[EG04] Anton M. Ertl and David Gregg. Retargeting JIT compilers by using C-compiler generated executable code. *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 13:41–50, 2004.

[EHFU13] Eva Eggeling, Andreas Halm, Dieter W. Fellner, and Torsten Ullrich. Optimization of an Autostereoscopic Display for a Driving Simulator. *Proceedings of the International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (GRAPP 2013)*, 8:318–326, 2013.

[Eic98] Jesse B. Eichenlaub. Lightweight compact 2D/3D autostereoscopic LCD backlight for games, monitor, and notebook applications. *Stereoscopic Displays and Virtual Reality Systems*, 5:180–185, 1998.

[EMP+02] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steve Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann, 2002.

[Euc07] Euclid. *Euclid's Elements of Geometry*. Fitzpatrick, Richard, 2007.

[Far90] Gerald Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press Professional, Inc., 1990.

[Far99] Gerald Farin. *NURBS for Curve and Surface Design from Projective Geometry to Practical Use*. AK Peters, Ltd., 1999.

[Far02] Gerald Farin. *Curves and Surfaces for CAGD: A Practical Guide*. Morgan Kaufmann Publishers Inc., 2002.

[FFBS04] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, Inc., 2004.

[FH05] Dieter W. Fellner and Sven Havemann. Striving for an adequate vocabulary: Next generation metadata. *Proceedings of the 29th Annual Conference of the German Classification Society*, 29:13–20, 2005.

[FH12] Gerald Frank and Christian Hillbrand. Automatic support of standardization processes in design models. *Proceedings of the International Conference on Intelligent Engineering Systems (INES)*, 16:393–398, 2012.

[Fin08] Dieter Finkenzeller. Detailed Building Facades. *IEEE Computer Graphics and Applications*, 28(3):58–66, 2008.

[Fla11] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly, Beijing Sebastopol, CA, 2011.

[FLS04] John Fisher, John Lowther, and Ching-Kuang Shene. If you know B-splines well, you also know NURBS! *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, 35:343–347, 2004.

[Fra11] Gerald Frank. Optimization of the Product Creation Process by Automated Design to Cost. *Proceedings of the International Conference on Intelligent Engineering Systems (INES)*, 15:363–367, 2011.

[FRC11] Donald L. Fisher, Matthew Rizzo, and Jeff K. Caird, editors. *Handbook of Driving Simulation for Engineering, Medicine, and Psychology*. Crc Press Inc., 2011.

[FT00] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *Proceedings of the 22nd International Conference on Software Engineering*, 22:407–416, 2000.

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice (2nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[Gar16] Julien Gardan. Additive manufacturing technologies: state of the art and trends. *International Journal of Production Research*, 54(10):3118–3132, 2016.

[GAVN11] Mohit Gupta, Amit Agrawal, Ashok Veeraraghavan, and Srinivasa G. Narasimhan. Structured Light 3D Scanning in the Presence of Global Illumination. *Proceedings of the 2011 IEEE Conference on Computer Vision and Pattern Recognition*, 1:713–720, 2011.

[GG06] Andreas Griesser and Luc Van Gool. Automatic Interactive Calibration of Multi-Projector-Camera Systems. *2006 Conference on Computer Vision and Pattern Recognition Workshop*, 1:8–15, 2006.

[GK07] Björn Ganster and Reinhard Klein. An Integrated Framework for Procedural Modeling. *Proceedings of Spring Conference on Computer Graphics 2007 (SCCG 2007)*, 23:150–157, 2007.

[GKP07] Eric J. Griffith, Michal Koutek, and Frits H. Post. Fast normal vector compression with bounded error. *Proceedings of the 5th Eurographics symposium on Geometry processing*, 5:263–272, 2007.

[GMTF89] Jack Goldfeather, Steven Monar, Greg Turk, and Henry Fuchs. Near Real-Time CSG Rendering Using Tree Normalization and Geometric Pruning. *IEEE Computer Graphics and Applications*, 9(3):20–28, 1989.

[GP07] Markus Gross and Hanspeter Pfister. *Point-Based Graphics*. Morgan Kaufmann Publishers Inc., 2007.

[GS69] Branko Grünbaum and Geoffrey C. Shephard. Convex Polytopes. *Bulletin of the London Mathematical Society*, 1(3):257–300, 1969.

[Guy17] Grard Guyard. Type3, a brand of Gravotech Marking. online: http://www.type3.com, 2017.

[Hav05] Sven Havemann. Generative Mesh Modeling. *PhD-Thesis, Technische Universität Braunschweig, Germany*, 1:1–303, 2005.

[HC04] Robin Hunicke and Vernell Chapman. AI for Dynamic Difficulty Adjustment in Games. *Proceedings of the Challenges in Game AI Workshop / Conference on Artificial Intelligence*, 19:91–96, 2004.

[HF04] Sven Havemann and Dieter W. Fellner. Generative Parametric Design of Gothic Window Tracery. *Proceedings of the 5th International Symposium on Virtual Reality, Archeology, and Cultural Heritage*, 1:193–201, 2004.

[HHKR97] Rolf Hammer, Matthias Hocks, Ulrich Kulisch, and Dietmar Ratz. *C++ Toolbox for Verified Computing*. Springer, 1997.

[Hig17] Jeff High. Gemvision corporation. online: http://gemvision.com/, 2017.

[HKS+10] Younis Hijazi, Aaron Knoll, Mathias Schott, Andrew Kensler, Charles Hansen, and Hans Hagen. CSG Operations of Arbitrary Primitives with Interval Arithmetic and Real-Time Ray Casting. *Scientific Visualization: Advanced Concepts*, 1:78–89, 2010.

[HKvdSV11] Mark Hills, Paul Klint, Tijs van der Strom, and Jurgen Vinju. A Case of Visitor versus Interpreter Pattern. *Proceedings of the International Conference on Objects, Models, Components and Patterns (TOOLS'11)*, 49:1–16, 2011.

[HL89] Josef Hoschek and Dieter Lasser. *Grundlagen der Geometrischen Datenverarbeitung (english: Fundamentals of Computer Aided Geometric Design)*. Teubner, 1989.

[HMVG09] Simon Haegeler, Pascal Müller, and Luc Van Gool. Procedural Modeling for Digital Cultural Heritage. *Journal on Image and Video Processing*, 9:1–11, 2009.

[HPTN06] Jukka Häkkinen, Monika Pölönen, Jari Takatalo, and Göte Nyman. Simulator sickness in virtual display gaming: a comparison of stereoscopic and non-stereoscopic situations. *Proceedings of the conference on Human-computer interaction with mobile devices and services*, 8:227–230, 2006.

[HRJS98] Scott T. Hemphill, Ilene M. Reinitz, Mary L. Johnson, and James E. Shigley. Modeling the appearance of the round brilliant cut diamond: An analysis of brilliance. *Gems & Gemology*, 34:158–183, 1998.

[HUF12] Sven Havemann, Torsten Ullrich, and Dieter W. Fellner. The Meaning of Shape and some Techniques to Extract It. *Multimedia Information Extraction*, 1:81–98, 2012.

[Int94] International Organization for Standardization (ISO) 10303-1:1994. Industrial automation systems and integration – Product data representation and exchange – Part 1: Overview and fundamental principles, 1994.

[Int08] International Organization for Standardization (ISO) 32000-1:2008. Document management – Portable document format – Part 1: Pdf1.7, 2008.

[Int12a] International Organization for Standardization (ISO) / Publicly Available Specification (PAS) 17506:2012. Industrial automation systems and integration – COLLADA digital asset schema specification for 3D visualization of industrial data, 2012.

[Int12b] International Organization for Standardization (ISO) 14306:2012. Industrial automation systems and integration – JT file format specification for 3D visualization, 2012.

[Int13] International Oganization for Standardization (ISO) / International Electrotechnical Commission (IEC) 19775-1:2013. Information technology – Computer graphics, image processing and environmental data representation – Extensible 3D (X3D) – Part 1: Architecture and base components, 2013.

[Int15] E. C. M. A. International. *ECMA-262: ECMAScript 2015 Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, sixth edition, 2015.

[Jer16] Jason Jerald. *The VR Book: Human-Centered Design for Virtual Reality*. Association for Computing Machinery and Morgan & Claypool, New York, NY, USA, 2016.

[JKIR06] Subramaniam Jayanti, Yagnanarayanan Kalyanaraman, Natraj Iyer, and Karthik Ramani. Developing an engineering shape benchmark for CAD models. *Computer-Aided Design*, 38:939–953, 2006.

[JMW+03] Adrian Jacobs, Jonathan Mather, Robert Winlow, David Montgomery, Graham Jones, Morgan Willis, Martin Tillin, Lyndon Hill, Marina Khazova, Heather Stevenson, and Grant Bourhill. 2D/3D Switchable Displays. *Sharp Technical Journal*, 4:1–5, 2003.

[Joh16] Andrew Johansen. *JavaScript: The Ultimate Beginner's Guide!* CreateSpace Independent Publishing Platform, 2016.

[JTSWF10] Martin Jennings-Teats, Gillian Smith, and Noah Wardrip-Fruin. Polymorph: A Model for Dynamic Level Generation. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 6:138–143, 2010.

[KBUF14] Patrick Knöbelreiter, René Berndt, Torsten Ullrich, and Dieter W. Fellner. Automatic fly-through Camera Animations for 3D Architectural Repositories. *Proceedings of the International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (GRAPP 2014)*, 9:335–341, 2014.

[KEL+15a] Ioana Koglbauer, Arno Eichberger, Cornelia Lex, Norbert Bliem, Anton Sternat, Jürgen Holzinger, Christoph Schinko, and Mario Battel. Bewertung von Fahrerassistenzsystemen von nicht professionellen Fahrerinnen und Fahrern im Realversuch. *Humanwissenschaftliche Beiträge zur Verkehrssicherheit und Ökologie des Verkehrs, mehr sicheres Verhalten im Strassenverkehr*, 5:86–102, 2015.

[KEL+15b] Ioana Koglbauer, Arno Eichberger, Cornelia Lex, Jürgen Holzinger, Christoph Schinko, and Torsten Ullrich. A Model for Subjective Evaluation of Automated Vehicle Control. *Proceedings of the International Symposium on Aviation Psychology*, 18:PW12, 2015.

[KEL+15c] Ioana Koglbauer, Arno Eichberger, Cornelia Lex, Jürgen Holzinger, Christoph Schinko, and Torsten Ullrich. Evaluation of driving maneuvers in reality and in an autostereoscopic 3D simulation with integrated eye-tracking. *Proceedings of the Human Factors and Ergonomics Society Europe Chapter 2015 Annual Conference*, 2015.

[KHH+07] Aaron Knoll, Younis Hijazi, Charles Hansen, Ingo Wald, and Hans Hagen. Interactive Ray Tracing of Arbitrary Implicits with SIMD

Interval Arithmetic. *Proceedings of IEEE Symposium on Interactive Ray Tracing*, 7:11–18, 2007.

[Khr14] Khronos WebGL Working Group. WebGL Specifcation. online: https://www.khronos.org/registry/webgl/specs/1.0/, 2014.

[Kit17a] Kitware, Inc. Paraview. online: http://www.paraview.org/, 2017.

[Kit17b] Kitware, Inc. The Visualization Toolkit (VTK). online: http://www.vtk.org/, 2017.

[KK11] Lars Krecklau and Leif Kobbelt. Procedural Modeling of Interconnected Structures. *Computer Graphics Forum*, 30:335–344, 2011.

[KLBL93] Robert S. Kennedy, Norman E. Lane, Kevin S. Berbaum, and Michael G. Lilienthal. Simulator Sickness Questionnaire: An enhanced method for quantifying simulator sickness. *International Journal of Aviation Psychology*, 7:203–220, 1993.

[Kob96] Leif Kobbelt. Interpolatory Subdivision on Open Quadrilateral Nets with Arbitrary Topology. *Computer Graphics Forum*, 15(3):409–420, 1996.

[KPK10] Lars Krecklau, Darko Pavic, and Leif Kobbelt. Generalized Use of Non-Terminal Symbols for Procedural Modeling. *Computer Graphics Forum*, 29:2291–2303, 2010.

[KSH+15] Hyosun Kim, Christoph Schinko, Sven Havemann, Ivan Redi, Andrea Redi, and Dieter W. Fellner. Tiled Projection onto Deforming Screens. *Computer Graphics and Visual Computing (CGVC)*, 1:35–42, 2015.

[KSHF13] Hyosun Kim, Christoph Schinko, Sven Havemann, and Dieter Fellner. Tiled Projection onto Bent Screens using Multi-Projectors. *Proceedings of the 7th IADIS International Conference on Computer Graphics, Visualization, Computer Vision and Image Processing*, 7:67–74, 2013.

[KSU14] Ulrich Krispel, Christoph Schinko, and Torsten Ullrich. The Rules Behind – Tutorial on Generative Modeling. *Proceedings of the 12th Symposium on Geometry Processing / Graduate School*, 12:2:1–2:49, 2014.

[KSU15] Ulrich Krispel, Christoph Schinko, and Torsten Ullrich. A Survey of Algorithmic Shapes. *Remote Sensing*, 7:12763–12792, 2015.

[KSU16] Ulrich Krispel, Christoph Schinko, and Torsten Ullrich. A Survey of Algorithmic Shapes. *Remote Sensed Data and Processing Methodologies for 3D Virtual Reconstruction and Visualization of Complex Architectures*, 219:498–529, 2016.

[KUF14] Ulrich Krispel, Torsten Ullrich, and Dieter W. Fellner. Fast and Exact Plane-Based Representation for Polygonal Meshes. *Proceeding of the International Conference on Computer Graphics, Visualization, Computer Vision and Image Processing*, 8:189–196, 2014.

[Lan11] Marcel Lancelle. Visual Computing in Virtual Environments. *PhD-Thesis, Technische Universität Graz, Austria*, 1:1–228, 2011.

[LD98] Bernd Lintermann and Oliver Deussen. A Modelling Method and User Interface for Creating Plants. *Computer Graphics Forum*, 17(1):73–82, 1998.

[LDMA+04] Johnny C. Lee, Paul H. Dietz, Dan Maynes-Aminzade, Ramesh Raskar, and Scott E. Hudson. Automatic Projector Calibration with Embedded Light Sensors. *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology*, 17:123–126, 2004.

[LHKR10] Douglas Lanman, Matthew Hirsch, Yunhee Kim, and Ramesh Raskar. Content-adaptive Parallax Barriers: Optimizing Dual-layer 3D Displays Using Low-rank Light Field Factorization. *ACM Trans. Graph.*, 29(6):163:1–163:10, 2010.

[Li17] Edmond Li. Jewellery CAD/CAM Limited. online: http://www.jcadcam.coms, 2017.

[LLL+15] Bo Li, Yijuan Lu, Chunyuan Li, Afzal Godil, Tobias Schreck, Masaki Aono, Martin Burtscher, Qiang Chen, Nihad Karim Chowdhury, Bin Fang, Hongbo Fu, Takahiko Furuya, Haisheng Li, Jianzhuang Liu, Henry Johan, Ryuichi Kosaka, Hitoshi Koyanagi, Ryutarou Ohbuchi, Atsushi Tatsuma, Yajuan Wan, Chaoli Zhang, and Changqing Zou. A comparison of 3D shape retrieval methods based on a large-scale benchmark supporting multimodal queries. *Computer Vision and Image Understanding*, 131(C):1–27, 2015.

[LNOM08] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro – The magazine for chip and silicon systems designers*, 28:39–55, 2008.

[Loo87] Charles Loop. Smooth Subdivision Surfaces Based on Triangles. *Master's Thesis, University of Utah, USA*, 1:1–74, 1987.

[Lor63] Edward N. Lorenz. Deterministic nonperiodic flow. *Journal of the Atmospheric Sciences*, 20(2):130–141, 1963.

[LWW08] Markus Lipp, Peter Wonka, and Michael Wimmer. Interactive Visual Editing of Grammars for Procedural Architecture. *ACM Transactions on Graphics*, 27:3:1–3:10, 2008.

[LWW10] Markus Lipp, Peter Wonka, and Michael Wimmer. Parallel Generation of Multiple L-Systems. *Computers & Graphics*, 34:585–593, 2010.

[LY99] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification*. Prentice Hall, 1999.

[LZQ06] Yi Liu, Hongbin Zha, and Hong Qin. Shape topics: A compact representation and new algorithms for 3d partial shape retrieval. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2:2025–2032, 2006.

[Mar98] George E. Martin. *Geometric Constructions*. Springer, 1998.

[Mig17] Migenius Pty Ltd. Migenius - Photorealistic Rendering in the Cloud Made Easy. online: http://www.migenius.com/, 2017.

[Mit90] William J. Mitchell. *The Logic of Architecture: Design, Computation, and Cognition*. MIT Press, 1990.

[MM08] Paul Merrell and Dinesh Manocha. Continuous Model Synthesis. *ACM Transactions on Graphics*, 27:158:1–158:9, 2008.

[MSH+08] Erick Mendez, Gerhard Schall, Sven Havemann, Dieter W. Fellner, Dieter Schmalstieg, and Sebastian Junghanns. Generating Semantic 3D Models of Underground Infrastructure. *IEEE Computer Graphics and Applications*, 28:48–57, 2008.

[MWH+06] Pascal Müller, Peter Wonka, Simon Haegler, Ulmer Andreas, and Luc Van Gool. Procedural Modeling of Buildings. *Proceedings of 2006 ACM Siggraph*, 25(3):614–623, 2006.

[MZWVG07] Pascal Müller, Gang Zeng, Peter Wonka, and Luc Van Gool. Image-based Procedural Modeling of Facades. *ACM Transactions on Graphics*, 28:3:1–3:9, 2007.

[NAM06] Mark J. Nelson, Calvin Ashmore, and Michael Mateas. Authoring an Interactive Narrative with Declarative Optimization-Based Drama Management. *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference*, 2:127–129, 2006.

[NAT90] Bruce Naylor, John Amanatides, and William Thibault. Merging bsp trees yields polyhedral set operations. *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, 24(4):115–124, 1990.

[Nic02] Franco Niccolucci. XML and the future of humanities computing. *ACM SIGAPP applied computing review*, 10(1):43–47, 2002.

[Nvi17a] Nvidia Corporation. Geforce Now - The New Way to Game. online: http://shield.nvidia.com/game-streaming-with-geforce-now, 2017.

[Nvi17b] Nvidia Corporation. Iray GPU Rendering. online: http://www.nvidia.com/object/nvidia-iray.html, 2017.

[Nvi17c] Nvidia Corporation. NVIDIA CUDA C Programming Guide. online: http://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2017.

[ÖK08] Mine Özkar and Sotirios Kotsopoulos. Introduction to shape grammars. *International Conference on Computer Graphics and Interactive Techniques ACM SIGGRAPH 2008 (course notes)*, 36:1–175, 2008.

[Par10] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 2010.

[PBP02] Hartmut Prautzsch, Wolfgang Boehm, and Marco Paluszny. *Bézier and B-Spline Techniques*. Springer, 2002.

[PKG+07] Tom Peterka, Robert L. Kooima, Javier I. Girado, Jinghua Ge, Daniel J. Sandin, and DeFanti Thomas A. Evolution of the Varrier Autostereoscopic VR Display: 2001–2007. *Stereoscopic Displays and Virtual Reality Systems*, 14:1–11, 2007.

[PL90] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.

[PL02] Helmut Pottmann and Stefan Leopoldseder. Geometries for CAGD. *Handbook of 3D Modeling*, 1:43–73, 2002.

[PM01] Yogi Parish and Pascal Mueller. Procedural Modeling of Cities. *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 28:301–308, 2001.

[Pol05] Tomas Polgar. *FREAX: The Brief History of the Computer Demoscene*. CSW-Verlag, Winnenden, 2005.

[PT97] Les Piegl and Wayne Tiller. *The NURBS book*. Springer-Verlag New York, Inc., 1997.

[PTY10] Christopher Pedersen, Julian Togelius, and Georgios N. Yannakakis. Modeling Player Experience for Content Creation. *IEEE Transactions on Computational Intelligence and AI in Games*, 2:54–67, 2010.

[QJS+06] Patrick Quirk, Tyler Johnson, Rick Skarbez, Herman Towles, Florian Gyarfas, and Henry Fuchs. RANSAC-Assisted Display Model Reconstruction for Projective Display. *Proceedings of the IEEE VR Workshop on Emerging Display Technologies*, 1:26–29, 2006.

[RBY+99] Ramesh Raskar, Michael S. Brown, Ruigang Yang, Wei-Chao Chen, Greg Welch, Herman Towles, W. Brent Seales, and Henry Fuchs. Multi-Projector Displays Using Camera-Based Registration. *Proceedings of the IEEE Conference on Visualization*, 1:161–168, 1999.

[RF07] Casey Reas and Ben Fry. *Processing: A Programming Handbook for Visual Designers and Artists*. MIT Press, 2007.

[RKT+12] Hayko Riemenschneider, Ulrich Krispel, Wolfgang Thaller, Michael Donoser, Sven Havemann, Dieter W. Fellner, and Horst Bischof. Irregular lattices for complex shape grammar facade parsing. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 25:1640–1647, 2012.

[RWC+98] Ramesh Raskar, Greg Welch, Matt Cutts, Adam Lake, Lev Stesin, and Henry Fuchs. The Office of the Future: A Unified Approach to Image-based Modeling and Spatially Immersive Displays. *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, 25:179–188, 1998.

[SBEF14] Christoph Schinko, René Berndt, Eva Eggeling, and Dieter Fellner. A Scalable Rendering Framework for Generative 3D Content. *Proceedings of the 19th International ACM Conference on 3D Web Technologies*, 19:81–87, 2014.

[Sch08] Uwe Schöning. *Theoretische Informatik - kurz gefasst.* Spektrum Akademischer Verlag, Heidelberg, 5th edition, 2008.

[SEFR14] Timo Scharwächter, Markus Enzweiler, Uwe Franke, and Stefan Roth. Stixmantics: A medium-level model for real-time semantic scene understanding. *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V*, 5:533–548, 2014.

[Set13] Volker Settgast. Processing Semantically Enriched Content for Interactive 3D Visualizations. *PhD-Thesis, Technische Universität Graz, Austria*, 1:1–233, 2013.

[SICH+14] Aitor Santamaria-Ibirika, Xabier Cantero, Sergio Huerta, Igor Santos, and Pablo G. Bringas. Procedural Playable Cave Systems based on Voronoi Diagram and Delaunay Triangulation. *Proceedings of the International Conference on Cyberworlds*, 12:15–22, 2014.

[SK92] John M. Snyder and James T. Kajiya. Generative modeling: a symbolic system for geometric modeling. *Proceedings of the 19th annual conference on computer graphics and interactive techniques*, 1:369–378, 1992.

[SKR+10] Kristian Sons, Felix Klein, Dmitri Rubinstein, Sergiy Byelozyorov, and Philipp Slusallek. Xml3d: interactive 3d graphics for the web. *Web3D '10: Proceedings of the 15th International Conference on Web 3D Technology*, 15:175–184, 2010.

[SKU08] László Szirmay-Kalos and Tamás Umenhoffer. Displacement mapping on the GPU - State of the Art. *Computer Graphics Forum*, 27(6):1567–1592, 2008.

[SKU15] Christoph Schinko, Ulrich Krispel, and Torsten Ullrich. Know the Rules – Tutorial on Procedural Modeling. *Proceedings of the 10th International Joint Conference on Computer Vision, Imaging and*

Computer Graphics Theory and Applications (GRAPP Tutorial Notes), 10:27ff, 2015.

[SKUF15] Christoph Schinko, Ulrich Krispel, Torsten Ullrich, and Dieter W. Fellner. Built by Algorithms – State of the Art Report on Procedural Modeling. *Proceedings of the 6th International Workshop on 3D Virtual Reconstruction and Visualization of Complex Architectures (3D-ARCH)*, 6:469–479, 2015.

[SMD+01] Daniel J. Sandin, Todd Margolis, Greg Dawe, Jason Leigh, and DeFanti Thomas A. The Varrier Auto-Stereographic Display. *Stereoscopic Displays and Virtual Reality Systems*, 8:1–8, 2001.

[Son17] Sony Corporation. Playstation Now - PS Now Subscription for PS3 Games. online: http://www.playstation.com/en-us/explore/playstationnow/, 2017.

[SPH+16] Christoph Schinko, Markus Peer, Daniel Hammer, Matthias Pirstinger, Cornelia Lex, Ioana Koglbauer, Arno Eichberger, Jürgen Holzinger, Eva Eggeling, Dieter W. Fellner, and Torsten Ullrich. Building a Driving Simulator with Parallax Barrier Displays. *Proceedings of the 11th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (GRAPP)*, 11:283–291, 2016.

[SPK+14] Ondrej Stava, Sören Pirk, Julian Kratt, Baoquan Chen, Radomir Měch, Oliver Deussen, and Bedrich Benes. Inverse Procedural Modelling of Trees. *Computer Graphics Forum*, 33:118–131, 2014.

[SS03] Daniel Scharstein and Richard Szeliski. High-accuracy Stereo Depth Maps Using Structured Light. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1:195–202, 2003.

[SSSS14] Helmut Schrom-Feiertag, Christoph Schinko, Volker Settgast, and Stefan Seer. Evaluation of Guidance Systems in Public Infrastructures Using Eye Tracking in an Immersive Virtual Environment. *Proceedings of the 2nd International Workshop on Eye Tracking for Spatial Research co-located with the 8th International Conference on Geographic Information Science*, 2:62–66, 2014.

[SSUF10a] Christoph Schinko, Martin Strobl, Torsten Ullrich, and Dieter W. Fellner. Modeling Procedural Knowledge – A Generative Modeler for Cultural Heritage. *Proceedings of the 3rd International Euro-Mediterranean Conference, (EuroMed)*, 6436:153–165, 2010.

[SSUF10b] Martin Strobl, Christoph Schinko, Torsten Ullrich, and Dieter W. Fellner. Euclides – A JavaScript to PostScript Translator. *Proceedings of the 1st International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking (Computation Tools)*, 1:14–21, 2010.

[SSUF11a] Thomas Schiffer, Christoph Schinko, Torsten Ullrich, and Dieter W. Fellner. Real-World Geometry and Generative Knowledge. *The European Research Consortium for Informatics and Mathematics (ERCIM) News*, 86:15–16, 2011.

[SSUF11b] Christoph Schinko, Martin Strobl, Torsten Ullrich, and Dieter W. Fellner. Scripting Technology for Generative Modeling. *International Journal On Advances in Software*, 4:308–326, 2011.

[Sta98] Jos Stam. Exact evaluation of catmull-clark subdivision surfaces at arbitrary parameter values. *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, 15:395–404, 1998.

[Sti80] George Stiny. Introduction to shape and shape grammars. *Environment and planning B*, 7(3):343–351, 1980.

[Str07] Bjarne Stroustrup. Evolving a language in and for the real world: C++ 1991-2006. *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages*, 3:4:1–4:59, 2007.

[SUF11] Christoph Schinko, Torsten Ullrich, and Dieter W. Fellner. Simple and Efficient Normal Encoding with Error Bounds. *Proceedings of Theory and Practice of Computer Graphics*, 29:63–66, 2011.

[SUF12] Christoph Schinko, Torsten Ullrich, and Dieter W. Fellner. Minimally Invasive Interpreter Construction – How to reuse a compiler to build an interpreter. *Proceedings of the 3rd International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking (Computation Tools)*, 3:38–44, 2012.

[SUF14] Christoph Schinko, Torsten Ullrich, and Dieter W. Fellner. Modeling with High-Level Descriptions and Low-Level Details. *Proceedings of the 8th International Conference on Computer Graphics, Visualization, Computer Vision and Image Processing*, 8:328–332, 2014.

[Sul04] Eldar Sultanow. Implizite Flächen. *Technical Report at Hasso-Plattner-Institut*, 1:1–11, 2004.

[SUSF11] Christoph Schinko, Torsten Ullrich, Thomas Schiffer, and Dieter W. Fellner. Variance Analysis and Comparison in Computer-Aided Design. *Proceedings of the 4th International Workshop on 3D Virtual Reconstruction and Visualization of Complex Architectures (3D-ARCH)*, 4:21–25, 2011.

[SVP+17] Christoph Schinko, Thomas Vosgien, Thorsten Prante, Tobias Schreck, and Torsten Ullrich. Search and Retrieval in CAD Databases – a user-centric State-of-the-Art Overview. *Proceedings of the 12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (GRAPP)*, 12:306–313, 2017.

[SYXH08] Weibin Sun, Xubo Yang, Shuangjiu Xiao, and Wencong Hu. Robust Checkerboard Recognition for Efficient Nonplanar Geometry Registration in Projector-camera Systems. *Proceedings of the 5th ACM/IEEE International Workshop on Projector Camera Systems*, 5:2:1–2:7, 2008.

[TDNL06] Julian Togelius, Renzo De Nardi, and Simon M. Lucas. Making Racing Fun Through Player Modeling and Track Evolution. *Proceedings of the Workshop on Adaptive Approaches for Optimizing Player Satisfaction in Computer and Physical Games*, 6:61–70, 2006.

[TDNL07] Julian Togelius, Renzo De Nardi, and Simon M. Lucas. Towards automatic personalised content creation for racing games. *IEEE Symposium on Computational Intelligence and Games*, 11:252–259, 2007.

[TKHF12] Wolfgang Thaller, Ulrich Krispel, Sven Havemann, and Dieter W. Fellner. Implicit Nested Repetition in Dataflow for Procedural Modeling. *Proceedings of the International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking (Computation Tools)*, 3:45–50, 2012.

[TKZ+13a] Wolfgang Thaller, Ulrich Krispel, René Zmugg, Sven Havemann, and Dieter W. Fellner. A Graph-Based Language for Direct Manipulation of Procedural Models. *International Journal on Advances in Software*, 6:225–236, 2013.

[TKZ+13b] Wolfgang Thaller, Ulrich Krispel, René Zmugg, Sven Havemann, and Dieter W. Fellner. Shape Grammars on Convex Polyhedra. *Computers & Graphics*, 37:707–717, 2013.

[TLL+11] Jerry O. Talton, Yu Lou, Steve Lesser, Jared Duke, Radomir Mech, and Vladlen Koltun. Metropolis Procedural Modeling. *ACM Transactions on Graphics*, 30:11:1–11:14, 2011.

[TMW02a] Robert F. Tobler, Stefan Maierhofer, and Alexander Wilkie. A Multiresolution Mesh Generation Approach for Procedural Definition of Complex Geometry. *Proceedings of the Shape Modeling International*, 8:35–44, 2002.

[TMW02b] Robert F. Tobler, Stefan Maierhofer, and Alexander Wilkie. Mesh-Based Parametrized L-Systems and Generalized Subdivision for Generating Complex Geometry. *International Journal of Shape Modeling*, 8:173–191, 2002.

[TYSB11] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-Based Procedural Content Generation: A Taxonomy and Survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3:172–186, 2011.

[UF11] Torsten Ullrich and Dieter W. Fellner. Generative Object Definition and Semantic Recognition. *Proceedings of the Eurographics Workshop on 3D Object Retrieval*, 4:1–8, 2011.

[Ull11] Torsten Ullrich. Reconstructive Geometry. *PhD-Thesis, Technische Universität Graz, Austria*, 1:1–322, 2011.

[U.S06] U.S. Product Data Association (US PRO), Formerly ANSI US PRO/IPO-100-1996. Initial Graphics Exchange Specification IGES 5.3, 2006.

[US13] Torsten Ullrich and Christoph Schinko. Bibliotheksdienste und semantische Auszeichnungen für digitale Artefakte. *Kulturelles Erbe in der Cloud – Fachtagung "Digitale Bibliotheken"*, 4:68ff, 2013.

[USB10] Torsten Ullrich, Volker Settgast, and René Berndt. Semantic Enrichment for 3D Documents: Techniques and Open Problems. *Publishing in the Networked World: Transforming the Nature of Communication, Proceedings of the International Conference on Electronic Publishing*, 14:374–384, 2010.

[USF08] Torsten Ullrich, Volker Settgast, and Dieter W. Fellner. Semantic Fitting and Reconstruction. *Journal on Computing and Cultural Heritage*, 1(2):1201–1220, 2008.

[USF10] Torsten Ullrich, Christoph Schinko, and Dieter W. Fellner. Procedural Modeling in Theory and Practice. *Poster Proceedings of the 18th WSCG International Conference on Computer Graphics, Visualization and Computer Vision*, 18:5–8, 2010.

[USSF13] Torsten Ullrich, Christoph Schinko, Thomas Schiffer, and Dieter W. Fellner. Procedural Descriptions for Analyzing Digitized Artifacts. *Applied Geomatics*, 5:185–192, 2013.

[VB07] Jan Vraný and Alexandre Bergel. The Debuggable Interpreter Design Pattern. *Proceedings of the International Conference on Software and Data Technologies*, 2:22–29, 2007.

[VGDA+12] Carlos A. Vanegas, Ignacio Garcia-Dorado, Daniel G. Aliaga, Bedrich Benes, and Paul Waddell. Inverse Design of Urban Procedural Models. *ACM Transactions on Graphics*, 31:168:1–168:11, 2012.

[VRM97] VRML Consortium, Inc. Information technology – Computer graphics and image processing – The Virtual Reality Modeling Language (VRML). online: http://tecfa.unige.ch/guides/vrml/vrml97/spec/, 1997.

[VV04] Emily A. Vander Veer. *JavaScript for Dummies*. For Dummies, 2004.

[Wal07] Ingo Wald. On fast construction of sah-based bounding volume hierarchies. *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, 1:33–40, 2007.

[Web08] Web3D Consortium, Inc. Information technology – Computer graphics and image processing – Extensible 3D (X3D). online: http://www.web3d.org/documents/specifications/19775-1/V3.2/, 2008.

[Wei07] Daniel Weiskopf. *GPU-Based Interactive Visualization Techniques*. Springer-Verlag Berlin Heidelberg, 1 edition, 2007.

[Wei17] Eric W. Weisstein. Spherical code. online: http://mathworld.wolfram.com/SphericalCode.html, 2017.

[WMG+07] Ingo Wald, William R. Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G. Parker, and Peter Shirley. State of the Art in Ray Tracing Animated Scenes. *STAR Proceedings of Eurographics*, 26:89–116, 2007.

[WYD+14] Fuzhang Wu, Dong-Ming Yan, Weiming Dong, Xiaopeng Zhang, and Peter Wonka. Inverse procedural modeling of facade layouts. *ACM Transactions on Graphics*, 33:121:1–121:10, 2014.

[YMYH12] Liu Yong, Zhang Mingmin, Jiang Yunliang, and Zhao Haiying. Improving procedural modeling with semantics in digital architectural heritage. *Computers & Graphics*, 36:178–184, 2012.

[YW01] Ruigang Yang and Greg Welch. Automatic and Continuous Projector Display Surface Calibration Using Every-Day Imagery. *Proceedings of 9th International Conf. in Central Europe in Computer Graphics, Visualization, and Computer Vision WSCG*, 9:320–327, 2001.

[YYT+11] Lap-Fai Yu, Sai-Kit Yeung, Chi-Keung Tang, Demetri Terzopoulos, Tony F. Chan, and Stanley Osher. Make it Home: Automatic Optimization of Furniture Arrangement. *ACM Transactions on Graphics*, 30:86:1–86:11, 2011.

[Zak11] Alon Zakai. Emscripten: An LLVM-to-JavaScript Compiler. *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, 1:301–312, 2011.

[ZL04] Dengsheng Zhang and Guojun Lu. Review of shape representation and description techniques. *Pattern Recognition*, 37:1–19, 2004.

[ZPKG02] Matthias Zwicker, Mark Pauly, Oliver Knoll, and Markus Gross. Pointshop 3D: an interactive system for point-based surface editing. *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, 29:322–329, 2002.

[ZWAY08] Jin Zhou, Liang Wang, Amir Akbarzadeh, and Ruigang Yang. Multi-projector Display with Continuous Self-calibration. *Proceedings of the 5th ACM/IEEE International Workshop on Projector Camera Systems*, 5:3:1–3:3, 2008.