René Zmugg - Institute of Computer Graphics and Knowledge Visualization

Procedural Creation of Man-Made Shapes and Structures

Towards Realizing a Digital Editable Representation of the World through Procedural Modeling

March 8, 2015

TU Graz

Graz University of Technology

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am ……………………………   ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

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 marked all material which has been quoted either literally or by content from the used sources.

……………………………   ………………………………………………..
date   (signature)

Abstract

The physical world is composed of three-dimensional objects. The usual way for people to digitize these environments is via photographs or videos, which map the three-dimensional world onto two-dimensional images and image sequences. In this momentary sampling process information is lost, which limits further use of this data. Tasks like measuring distances, comparing sizes, or viewing objects from other directions are simply not possible in a two-dimensional representation. In contrast, within a static three-dimensional representation, information loss is reduced and these tasks become possible, but this representation still remains a momentary snapshot of the physical world. By further incorporating semantic information, which includes, among others, symmetries and repetitions, the information loss can be reduced to a minimum and variations of the recorded object can be visualized. Unfortunately, acquiring these superior three-dimensional representations is a much harder task.

Nevertheless, three-dimensional content is already used in a great variety of applications. From motion pictures and games, to websites and PDF documents, three-dimensional data has been integrated in all of them. Three-dimensional data is used to show variations of reality and make complex situations understandable in an intuitive way. This data, however, is usually generated by specialized experts and not all established tools provide an easy way to generate three-dimensional content data for the regular user. This creates a hurdle for common users to create such content for their own purposes. For them to utilize three-dimensional content creation efficiently, simplified and more accessible tools are needed.

As mentioned before, it is important what kind of content is created to reduce the information loss. While static content is well-suited for applications that do not require any changes of the model, models that are used for example in planning tasks, mass generation, or changing environments need to encapsulate semantic information to provide a proper editable description. Especially in models of man-made shapes, this information can be exploited to generate such descriptions, which allow generation of a whole family of related shapes. The key to accomplish this is the understanding of shape. This is, however, no trivial task.

The creation of editable object descriptions is one possible way to allow everyone to create three-dimensional content. This vision motivates research to reach an *editable*, digital representation of the physical world, which will simplify the use and generation of three-dimensional data. To implement this in a feasible way, techniques beyond laser scanning and manual modeling are necessary. One very promising way to realize this is *procedural modeling*.

The world, however, is so vast and versatile that describing all object categories in the context of one thesis is simply not feasible, but, fortunately, a lot of experience on procedural modeling has been available in our research group. Thus, this thesis can take a first step on the road to realize this vision by exploring representative, sufficiently complex and comprehensive problems in the domain of man-made shapes. These problems are mainly analyzed on an application level to identify requirements for developed methods in an early step. The problems analyzed in this thesis can be grouped into three major problem areas, which have been carefully selected. These are: the understanding of shape, the reconstruction of architecture, and interactive procedural modeling. I developed new techniques in these areas to ease and guide the construction of procedural models for everyone. These techniques include a generic method for creating procedural models from a set of exemplars, improvements of the state of the art of split grammars in the domain of procedural architecture, and methods for interactive creation of animated procedural environments.

Kurzfassung

Unsere reale Welt besteht aus dreidimensionalen Objekten. Die für Menschen übliche Art und Weise, Teile dieser Welt zu digitalisieren, ist das Aufnehmen von Fotos oder Videos. Diese Techniken bilden die dreidimensionale Welt auf zweidimensionale Bilder und Bildersequenzen ab. Durch diesen Prozess werden nur Momentaufnahmen gewonnen und dadurch wird Information verworfen. Dies limitiert die weitere Verwendung dieser Daten. In einer zweidimensionalen Abbildung sind Tätigkeiten wie Abstandmessen, Größenvergleiche oder Betrachten von anderen Blickwinkeln nicht möglich. Eine statische dreidimensionale Repräsentation dieser Daten ist hingegen von weniger Verlusten behaftet und die zuvor erwähnten Tätigkeiten können durchgeführt werden. Es handelt sich jedoch weiterhin nur um eine Momentaufnahme. Durch das zusätzliche Einbetten von semantischen Informationen, wie zum Beispiel Symmetrien oder Wiederholungen, kann der Verlust an Information auf ein Minimum reduziert werden und Variationen des aufgenommenen Objekts können visualisiert werden. Unglücklicherweise sind diese überlegenen dreidimensionalen Repräsentationen viel aufwendiger zu generieren.

Nichtsdestotrotz werden dreidimensionale Inhalte bereits vielseitig eingesetzt. Einsatzgebiete beinhalten zum Beispiel Filme, Videospiele, aber auch Webseiten und PDF-Dokumente. Dreidimensionale Daten werden verwendet, um Variationen der Wirklichkeit zu visualisieren und komplexe Sachverhalte verständlich darzustellen. Üblicherweise werden diese Daten von spezialisierten Experten erstellt. Für nicht spezialisierte Benutzer hingegen bieten die Software-Werkzeuge, die in diesem Prozess verwendet werden, keinen einfachen Weg, solche Daten zu generieren. Diese Hürde erschwert diesen Benutzern die Generierung von dreidimensionalen Daten für ihre eigenen Zwecke. Einfachere und leichter zugängliche Werkzeuge sind notwendig, um diesen Benutzern die effiziente Erstellung solcher Daten zu ermöglichen.

Wie zuvor erwähnt, ist es wichtig, den richtigen Typ an dreidimensionalen Inhalten zu generieren, um den Informationsverlust gering zu halten. Statischer, unveränderlicher Inhalt ist gut geeignet für Anwendungen, in denen keine Änderungen an den Modellen vorgenommen werden müssen. Modelle, die unter anderem für Planungen, Massengenerierung oder veränderliche Umgebungen genutzt werden, müssen semantische Informationen beinhalten, um eine angemessene editierbare Beschreibung zu ermöglichen. Besonders für Objekte, die von Menschenhand geschaffen wurden, kann diese Information ausgenutzt werden, um solche Beschreibungen zu generieren. Unter Einhaltung der semantischen Bedingungen erlauben es diese Beschreibungen, eine ganze Familie an verwandten Modellen zu erzeugen. Der Schlüssel für all dies ist das Formverstehen. Diese Aufgabe ist jedoch kein einfaches Unterfangen.

Die Erstellung von editierbaren Objektbeschreibungen ist ein möglicher Weg durch den jedem das Erzeugen von dreidimensionalen Inhalten zugänglich gemacht wird. Diese Vision motiviert die Forschung, eine *editierbare*, digitale Beschreibung der physischen Welt am

Computer zu erreichen, um die Verwendung und Generierung dreidimensionaler Daten zu vereinfachen. Für eine realistische Umsetzung dieser Vision sind Techniken notwendig, die über Scanning und manuelles Modellieren hinaus gehen. Ein sehr vielversprechender Weg, dies zu realisieren, ist *prozedurales Modellieren*.

Die Welt ist jedoch so groß und vielfältig, dass das Beschreiben aller Objektkategorien im Kontext einer Dissertation nicht realisierbar ist. Glücklicherweise besteht viel Erfahrung im Bereich prozedurales Modellieren in unserer Forschungsgruppe. Daher kann diese Arbeit, durch das Untersuchen von repräsentativen, ausreichend komplexen und reichhaltigen Problemen, basierend auf von Menschen geschaffenen Objekten, den ersten Schritt, um diese Vision umzusetzen, tätigen. Diese Probleme werden in dieser Arbeit größtenteils auf Anwendungsebene analysiert, um Anforderungen für die entwickelten Techniken früh zu erkennen. Die untersuchten Probleme lassen sich in drei Hauptgruppen gliedern, welche sorgfältig ausgewählt wurden. Diese sind Formverstehen, Rekonstruktion von Architektur und interaktives prozedurales Modellieren. Ich habe neue Techniken in diesen Bereichen entwickelt, um den Konstruktionsprozess von dreidimensionalen Modellen zu vereinfachen und den Benutzer dabei zu unterstützen. Diese Techniken beinhalten eine generische Methode, um prozedurale Modelle aus einer Menge von Exemplaren zu erstellen, sowie Verbesserungen des Stands der Technik von Split-Grammatiken im Bereich der prozeduralen Architektur und Methoden zur interaktiven Generierung von animierten prozeduralen Umgebungen.

Acknowledgements

I would like to thank my supervisor Sven Havemann and the head of the Institute of Computer Graphics and Knowledge Visualization Dieter W. Fellner for guiding me through writing this thesis. Furthermore, thanks to all my colleagues at the institute and Fraunhofer Austria at Graz University of Technology for their support. I especially want to thank Wolfgang Thaller, Ulrich Krispel and Johannes Edelsbrunner with whom I had a close collaboration in our shared fields of research. Their cooperation and continuous support had a significant impact on my work. Furthermore, I want to express my gratitude to all my other co-authors, who published scientific articles in the context of our work. Finally, also thanks to my family and friends, who supported me in this time of writing.

Publications

The majority of the contributions and results of this thesis have been – respectively will be – published in the following articles and conference proceedings:

[ZKT*14] Zmugg R., Krispel U., Thaller W., Havemann S., Pszeida M., Fellner D. W.. A new approach for interactive procedural modelling in cultural heritage. *In* Archaeology in the Digital Era: Papers from the 40th Annual Conference of Computer Applications and Quantitative Methods in Archaeology (CAA 2012), 40:190 – 204, 2014.

[ZTH*12] Zmugg R., Thaller W., Hecher M., Schiffer T., Havemann S., Fellner D. W.. Authoring animated interactive 3D museum exhibits using a digital repository. *In* The 13th International Symposium on Virtual Reality, Archaeology and Cultural Heritage (VAST 2012), 13:73 – 80, 2012.

[TZK*13] Thaller W., Zmugg R., Krispel U., Posch M., Havemann S., Fellner D. W.. Creating procedural window building blocks using the generative fact labeling method. *In* ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, XL-5/W1:235 – 242, 2013.

[TKZ*13a] Thaller W., Krispel U., Zmugg R., Havemann S., Fellner D. W.. A Graph-Based Language for Direct Manipulation of Procedural Models. *In* International Journal On Advances in Software, 6/3&4:225 – 236, 2013.

[TKZ*13b] Thaller W., Krispel U., Zmugg R., Havemann S., Fellner D. W.. Shape grammars on convex polyhedra. *In* Computers & Graphics, 37/6:707 – 717, 2013.

[ZTK*13] Zmugg R., Thaller W., Krispel U., Edelsbrunner J., Havemann S., Fellner D. W.. Deformation-aware split grammars for architectural models. *In* 2013 International Conference on Cyberworlds (CW 2013), 12:4 – 11, 2013.

[ZTK*14] Zmugg R., Thaller W., Krispel U., Edelsbrunner J., Havemann S., Fellner D. W.. Procedural Architecture using Deformation-Aware Split Grammars. *In* The Visual Computer, 30/9:1009 – 1019, 2014.

[BCZ*14] Braun A., Cieslik S., Zmugg R., Wichert R., Klein P. and Havemann S.. V2me – Virtual Coaching for Seniors. *In* Wohnen – Pflege – Teilhabe – Besser leben durch Technik. 7. Deutscher AAL-Kongress, 2014.

Contents

Chapter 1
Introduction

Contents

Abstract. The vision for everyone to process and generate three-dimensional content to realize their tasks directly leads to the desire to reach a digital and editable representation of the real world on the computer. This ambition has been formulated as the goal of the research field visual computing, which combines the fields of computer graphics and computer vision. Through a comparison with other established modeling techniques, this section motivates that procedural modeling is a viable approach for the realization of this goal. The world, however, is so vast and versatile that its whole extent cannot be captured in the context of one thesis. Therefore, to take the first step in this direction, this thesis inspects a variety of representative problems concerning the generation of procedural models of man-made shapes. These problems can be appointed to three major problem fields, which include the problem of understanding shape spaces, the reconstruction of architecture, and interactive procedural modeling.

1.1 Motivation

The world around us is composed of three-dimensional objects and humans are used to interact with these objects in a natural way. However, data that is taken from this world and transferred to computers is mainly two-dimensional. This is mostly due to the fact that two-dimensional data, like a photograph, is easier to acquire. However, this two-dimensional data is only a momentary projection of the three-dimensional truth, and information is lost in this sampling process. This loss of information is of course acceptable for certain applications, such as portrait photos, but without this complete information further use is limited. It is, for example, not possible to generate further views of the photographed object, even simple tasks like measuring distances becomes impossible in the two-dimensional representation. As an example, take a photograph of a façade from a street. The single photograph provides a single view, from which relations between sizes can be estimated, but direct measurements are impossible. Furthermore, a lot of information is not available because of objects occluding one another. In a static three-dimensional representation of this façade, which is still just a momentary snapshot of the world, distances and sizes of elements can be measured directly and the façade can be looked at from arbitrary directions to reveal more information. However, to reduce the information loss to a minimum, semantic information – such as repetitions, symmetries and orthogonalities – need to be embedded into this representation. This allows further use of this data, e.g. using this façade with different proportions in a virtual street. Unfortunately, such representations are much harder to acquire than simple photographs.

Nevertheless, for the last twenty years, three-dimensional content has been continuously and successfully integrated all around us. Computer graphics are gradually spreading over from specialized domains into applications for the regular user, but for them to efficiently create their own three-dimensional content, work has still to be done. We can observe that websites increasingly rely on three-dimensional data for visualizing their content. The ability to display three-dimensional content has also been introduced to the PDF file format that is widely used. The smart phone trend has also revealed a new market for three-dimensional applications. Geographical applications like Google Maps [Goo13b] have reached an important position, especially on mobile platforms, and are now shifting from two-dimensional representations to three-dimensional ones. This *paradigm shift* indicates that three-dimensional structures and shapes are understood intuitively if they are presented in the correct way. However, there are still established software tools such as Microsoft Office [Mic13c], which do not incorporate three-dimensional content in an efficient way. Especially in a presentation software like Microsoft Powerpoint, displaying three-dimensional content would be desirable in many circumstances. Nevertheless, by following this trend it becomes possible that three-dimensional content will be displayed and available everywhere in the future.

However, the problem of how regular users can create their own three-dimensional content still remains. There are many applications that concern altering the real world, such as furnishing and planning the own home. It is not always possible and feasible to carry out such tasks in the real world from the start. To realize work like this efficiently, support from three-dimensional software is needed. These planning and modeling tasks are usually carried out by specialized experts. The tools these experts use are available for everyone, but are too specialized to appeal and be accessible to everyone. However, especially tasks like furnishing an apartment is something that many people like to do on their own. For this, means to create appropriate three-dimensional content are necessary.

Independent from the way the content is created, it is important that the loss of information is kept low. Thus, the generated content should not be static, but editable. Static content, like a model acquired by a laser scanner, cannot adapt to changing environments. In the example of furnishing the own home, three-dimensional cupboard models should scale to fit the available space by introducing additional drawers and shelves. To efficiently generate an editable three-dimensional representation of a real world (man-made) object, inherent properties that users want to exploit should be represented explicitly. This semantic information prevents inconsistencies in the model and allows modifications and the reuse of parts in a consistent way. This allows an efficient generation of a whole family of similar and related shapes that is represented by one common description. Furthermore, the data necessary to describe the elements of this shape family is reduced by a substantial amount. Incorporating this semantic information to enable a proper editable representation, however, requires understanding of the underlying shapes, which is not an easy task.

The potential that lies within three-dimensional content is huge, but by far not exhausted and utilized in a way that fulfills this potential. Three-dimensional content allows the modification of the world inside the computer to visualize different hypotheses without realizing them first in the real world. However, with the technological standpoint we have today it is not yet possible to process three-dimensional content with the same ease as two-dimensional content. Consequently, the technology is not developed far enough for everyone to process three-dimensional data, but by providing editable object descriptions, three-dimensional content creation can be made more accessible. This vision for everyone to process three-dimensional data motivates research to reach an *editable* three-dimensional representation of the physical world on the computer, which in turn will lead to new techniques that simplify the use and creation of three-dimensional content.

1.2 Visual Computing and its Long-Term Vision

Parallel to computer graphics, which mainly focuses on *image synthesis*, is the field of computer vision for *image analysis*. Those two fields are viewed as separate domains, but are closely related to each other. Computer graphics traditionally solve *forward problems*, such as rendering a three-dimensional scene to a two-dimensional image, or creating or editing digital representation of shapes and surfaces. This way three- or multi-dimensional data is conveyed to human beings. Computer vision on the other hand focuses more on *inverse problems*, like producing a three-dimensional digital representation out of two-dimensional image data. Approaches in one field can profit from approaches from the other field. The relations between both fields can be understood as traversing from three-dimensional data to two-dimensional data and back (see Figure 1.1). The research field that emerged through the combination of both is called *visual computing*.

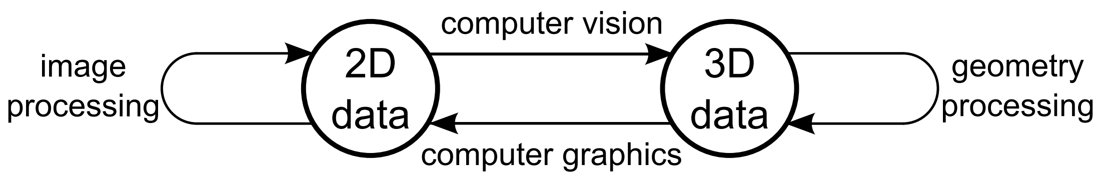


Fig. 1.1 Overview of the domain of visual computing. The research fields of computer vision and graphics are closely related to each other. Three-dimensional data is rendered to two-dimensional images in computer graphics and two-dimensional images are analyzed to create three-dimensional data in computer vision. (image adapted from Fellner [Fel92])

As a result of this ongoing convergence of the fields of computer graphics and computer vision, Hans Peter Seidel, the head of the computer graphics department of the Max Planck institute in Saarbrücken, Germany, prominently formulated during his Leibniz award speech in 2003 that the goal of visual computing is to generate an editable digital representation of the real physical world. So, the field of visual computing can be understood as to connect the real world with its corresponding digital counterpart. To fulfill the aforementioned goal, there are several ways to map the real world to a digital representation, but what techniques are really feasible to describe the whole physical world?

Three-Dimensional Laser Scanning.

Through the introduction of the Microsoft Kinect Sensor [Mic13b] scanning hardware has become available for a reasonable price on the market. Laser scanning is a way how the world can be sampled. Just like audio signals, a surface can be understood as a signal that can be sampled. With a higher sampling frequency a more detailed scanned model can be obtained. Scanning, however, does not produce a proper editable representation. Planar surfaces, for example, are represented by a vast number of points, even though a small set of points would suffice. In terms of being an editable representation there are too many degrees of freedom because each single point can be altered. This gives the user control over position of all vertices, but due to the missing semantics there is no control for high-level editing of the model. Furthermore, through scanning it is only possible to digitize objects that already exist and it is thus mostly suitable for documentation purposes. Just imagine a scanned representation of a chair. A reasonable editing operation would be varying the width of the chair, which in turn influences the width of the backrest as well as the position of the legs of the chair. Applying this change to a scanned chair is very costly. To enable such changes in an easy way, a procedural description is more appropriate than a scanned model. With just a small amount of parameters, a procedural model can describe a set of different chairs more efficiently than individual scans, where each has millions of surface points.

Scanning is useful for obtaining digital representations of objects that need no change or editing, for example artifacts for museum exhibitions, but the world is just too vast and manifold to consider scanning as a viable alternative for the goal of visual computing.

Manual Modeling.

The approach of manual creation of geometric models does not suffer from the shortcomings of the scanning approach. Manual modeling gives total control over the modeling process, so, for example, planar surfaces can be efficiently represented by as many points as are necessary. Therefore, the editability is not as restricted as with the scanning approach. However, in terms of editing, the possibilities are limited. Early modeling steps may not accessible any more in the later modeling process. Furthermore, the generated shapes contain no semantic information and are only based on geometric primitives. Post processing steps like mesh simplification may destroy invariants like symmetry because of the lack of semantics. A single triangle, which is part of a triangle soup, simply has no information to which part of the model it belongs. The main issue is, above all, the effort necessary to create models manually. The amount of modeled details is directly proportional to the amount of used modeling operations. The hours needed to create a model define the model's price. The time needed to create a single fairly complex model may be too high to consider making different variations of this model. Considering the amount of highly detailed geometric models that are needed for feature movies and computer games, a purely manual creation process is no longer feasible.

Structured Shape Editing.

The main focus of structured shape editing lies within structure-preserving shape manipulations. Within the space of man-made shapes structural invariants are, for example, symmetry, co-planarity, orthogonality, or regular arrangements. Novel shapes are either created by editing an input shape while maintaining the structure or by assembling parts from shapes out of an existing collection while retaining the initial structure. Through the latter – so-called *data-driven* methods – existing three-dimensional models (from on-line repositories such as Trimble 3D Warehouse [Goo13d]) can be reused and combined to create entirely new shapes and span a design space. However, to recombine parts automatically, a viable segmentation of different reusable parts is necessary.

However, there are two main drawbacks of this method. On the one hand the basic population of all shapes cannot be learned by machine learning techniques. So far only very small sets have been automatically analyzed, which leads to a very small freedom in variability. In many situations multiple examples to learn from are not available and learning approaches based on single instances are not productive. The second drawback from reusing pre-modeled parts is the artifacts that come from non-uniform scaling. Scaling parts non-uniformly leads to distortion of these. This way, for example, circular segments become elliptical, which is not always desirable. For a correct adaptation to scaling, the parts need to be described entirely, which is not possible with automatic methods. Structured shape editing methods are discussed in more depth in Section 2.6.

Procedural Modeling.

Shapes should react to scaling appropriately and should exhibit great variability to describe shapes from a common family. To realize this, a *procedural approach* is the most appropriate alternative. Look, for example, at the images in Figure 1.2. They depict different objects from chairs to beds, which, however, all share common features. In comparison to parametric modeling, which only allows changing of parameters, procedural modeling also allows complete structural changes as seen in the different designs of the beams between the chair legs. It is important to extract the essential information out of similar objects. This information is represented in the differences between these objects, in case of the example before, differences lie for example in the height of the legs, area between the legs, or the angle of backrest. To find a shared parametrization of similar objects, each of the objects in question has to become a marginal case or rather a special case of this parametrization. Interpolation – and in some cases extrapolation – of the parameter values, that describe models, yield further valid results.

The main problem with procedural techniques lies in the conceptual complexity of the underlying task: *shape understanding*. Furthermore, shapes are, ideally, represented through a description of minimal length providing only the essential degrees of freedom. The Kolmogorov complexity $KC(b)$ of a bit sequence b is the length of the shortest computer program that (re-)produces b. This measure, however, is not computable. Consequently, it is not possible to say for sure that the shortest procedural description has been found. Properties as well as drawbacks of the procedural approach are discussed in detail in the upcoming Section 1.3.

Visual Computing, therefore, can be seen as the sum of procedural approaches in computer graphics and computer vision. We can, furthermore, expand visual computing's ideal by taking our imagination into account. There is far more we can imagine than what actually exists. Revolutionary movies such as *The Matrix* (1999), or *Minority Report* (2002) inspired research in several fields. Video games show us the same, things that do not yet exist, or will simply never exist, but can be conceived by the human mind. We should therefore reformulate the

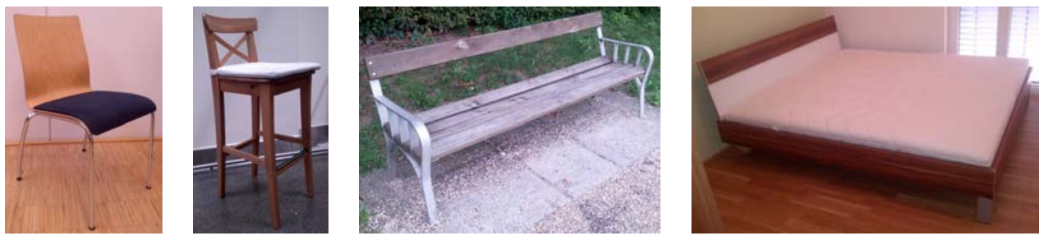


Fig. 1.2 A family of objects that are similar to chairs. All of these objects share similar features and can therefore be characterized by a single procedural description with different parameters. Each object can then be represented by a parameter assignment. Interpolation – and in some cases extrapolation – of these parameter values will yield further objects of this family. Structural changes that are not encoded in a basic model, such as the beams between the chair legs, are not possible with parametric modeling alone, procedural modeling is necessary to achieve such changes.

goal of visual computing to map *the magnitude of things we can imagine* to an editable digital representation. This, however, implies that techniques need to become more flexible to allow describing objects and shapes that we can only imagine so far.

Nevertheless, the world, as well as our imagination, is so vast and versatile, that it is hard to capture its whole extent. There is no real starting point for this endeavor, but we have to start somewhere; therefore, the first step is taken by inspecting a variety of representative problems in the domain of man-made shapes in scope of this thesis.

1.3 The Promise of Procedural Modeling

A procedural model is no three-dimensional object or set of objects itself, it is the program that describes the blueprint of these objects and generates them. Procedural modeling means creating things by coupling several procedures together. These procedures – small programs with a well-defined input and output – are executed in a clearly defined order.

The advantages of procedural modeling are manifold when compared to traditional manual modeling:

- **High-Level Representation:** With appropriate procedures it is not necessary to resort to low-level operations, such as moving vertices. In many cases only a few essential degrees of freedom are required to describe the essence of a shape. These degrees of freedom, however, are always dependent on the interpretation
- **Database Amplification:** Procedural models have a very small memory footprint compared to the data of the model they produce. Therefore, it is not necessary to store the full data of the produced outputs. Highly complex models can be acquired from a relatively small set of construction rules.
- **Variability:** Procedural models are versatile and variable. Arbitrary many variations of one prototype object can be generated by changing the parameters and by using the same set of rules.
- **Semantic Description:** Entirely different models and entire model families can be generated by one procedural model. Once the high-level structure is understood a lot of special cases can be described.
- **Level of Detail:** With the appropriate rule design, different level of detail can be produced for any procedural model.

To utilize procedural modeling techniques, planning is very important. You have break down the tasks into manageable procedures and have to define the rules and steps, together with the corresponding parameters. The final outcome, however, is never specified directly. It is important that the exact same result can always be reproduced with the procedures and parameters available. In contrast, for the case of traditional manual modeling this is not the case. Here craftsmanship skill is required instead of a logical mind and careful planning. The exact same result cannot be reproduced, not even by the same person.

In general, for procedural modeling much more planning effort is necessary, but the amount of time for building and fine-tuning is less than for manual modeling. However, for each additional model from the same family you want to represent, the time needed for planning is reduced. Observations from the objects addressed before can be reused to describe further objects. Manual mesh modeling is like a complement to procedural modeling, the planning effort is minimal, but a lot of time is spent in the modeling and fine-tuning part. Figure 1.3 illustrates this comparison.

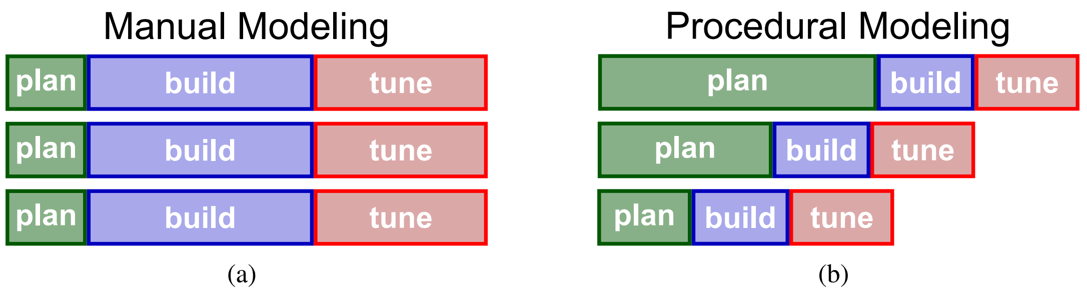


Fig. 1.3 With manual mesh modeling (a) the planning effort is minimal, but a lot of time is spent during building and fine-tuning of the model. No benefit is gained by modeling more similar models. With procedural modeling (b) there is a big initial planning effort, however, because of the similarity of the related objects the planning effort for further objects is decreased and the overall time needed is reduced.

The potential of procedural modeling approaches is immense. However, it is not easy to make use of all the mentioned advantages. There are two important application areas that utilize procedural modeling approaches in a big way:

Product Development. There is no room for traditional modeling in the field of product development; each model features a lot of different parameters to describe its shape. These parameters are fine-tuned until all constraints are met and simulations are satisfactory. The main tool here is computer-aided design software, which is used in all stages of product development from conceptualization and design to manufacturing and engineering.

Video Games and Movies. The scale of video games and movies today is getting bigger and bigger. The amount of three-dimensional models in domains like vegetation and urban structures, for which satisfactory detail is produced, is so high that it is not feasible to create all these models manually anymore. The positioning of all these models is also guided by procedural techniques.

However, a very important question remains: Can everything be described procedurally? (discussed in detail in Section 7.1) The abundance of shapes within object families sometimes feel endless and not parametrizable, which leads to difficulties in mapping the real underlying structure of a family of objects into a procedural description. This is also due to problems in understanding or finding the underlying structures. Hope in this regard lies within *factorization*. One example here is the universal applicability of profiles on man-made shapes. Sweeping of profiles is an important modeling tool with a lot of variability and expressiveness. By factoring out profiles, complicated parametrization problems can suddenly become manageable. For example, by removing profiles from windows, pillars, and other façade elements. In many cases only simple geometric entities remain that can be easily described by procedural means. Hope here is that there are a lot of similar operations that can be factored out, so that the problems get smaller and a whole parametrization can be found. There is, however, no proof that factorization is the solution to overcome the huge variety within man-made objects, but it is at least a way to ease the problem. In general, as long as rules to describe a shape exist it is possible to exploit them to reach a procedural description.

1.4 Representative Problems

The field of applications for procedural techniques is huge, but to explore the advantages and limitations of procedural techniques – in particular procedural modeling – different representative problems in the field of the reconstruction of man-made shapes and structures will be examined in this thesis. Structural symmetry lies in the nature of man-made shapes; almost all man-made shapes contain parts that are similar in shape, style, detail, function or purpose. These observations are very important to generate procedural descriptions of these shapes.

This thesis presents complex problems that are suitable for the procedural approach. Applied research allows to identify requirements that remain hidden in a theoretical approach. To fulfill a majority of these requirements, I will analyze most of the the discussed problems on an application level. In this introductory part, all problems that are discussed in this thesis are grouped into three major problem areas, which are discussed in the following three sections. These representative areas are: the understanding of shape (see Section 1.4.1), the reconstruction of architecture (see Section 1.4.2), and interactive procedural modeling (see Section 1.4.3).

Collaborations in the Field of Procedural Modeling. Even though I have dedicated my work to these three areas, these large domains contain difficult problems, which are hard to work on alone. Besides developing new solutions, I have also used technologies that existed or are in development at our institute. This, of course, leads to tight collaborations with my colleagues at the institute. I, especially, did a lot of my work in cooperation with Krispel [Kriar] and Thaller [Thaar]. As we all work in the same domain, we share the ambitions for developing new and improved ways of procedural modeling and pursue the goal of understanding shapes.

My focus of research lies in application-level procedural modeling, where I concentrate myself on the application and practical suitability of the developed techniques. I explore to what degree the techniques can be used, what the limitations are and what extensions are needed on the application-level. For this cause, I developed extensions and improvements for established techniques. Krispel's focus is in the field of geometry processing for procedural modeling. In context of his research he developed a robust modeling vocabulary based on convex polyhedra, which has been extensively used by me in the domain of procedural reconstruction of architecture. Thaller concentrates his research on the concepts of language and structure within procedural modeling. He developed the foundation for an interactive procedural modeling tool named the GML Compositor. I used and extended this tool for projects in the domain of interactive procedural modeling.

1.4.1 Understanding Shape Spaces

Procedural models are often obtained from analyzing a set of exemplars, which span an entire *shape space*. The exemplars serve as marginal cases for the procedural model and through interpolation of these new shapes can be derived from the procedural description. The acquired procedural models can be used to fit a three-dimensional model to real life objects. This field is known as *Generative Surface Reconstruction* (see Section 2.4.1).

The foundation for formulating a shape space lies within *inductive* reasoning. In comparison to *deductive* reasoning, which takes general statements to make assumptions about something special, inductive reasoning tries to make general assumptions based on observations of single special elements. Naturally, inductive reasoning is the much harder task, but this is exactly what is necessary to obtain a procedural description. Based on the idea of inductive reasoning, the Generative Fact Labeling method, which is demonstrated in Section 5.2.1, has been developed by our research team to organize the process to reach a procedural description based on a set of given exemplars.

Shapes spaces can either be *closed* or *open*. One example for a closed shape space would be the space spanned by wedding rings available from a single manufacturer. Throughout these rings, all available features are demonstrated, and the recombination of those creates entirely new shapes that are within the manufacturing capabilities. This field is comprehensive enough to demonstrate the way to formulate a shape space and this is demonstrated in Section 6.1.

An open shape space is, for example, spanned by the multitude of different architectural elements that occur often in slightly varied ways in other pieces of architecture. These parts include, among other things, windows, doors, pillars, and staircases. Creating procedural models of such elements is very advantageous in terms of re-usability. In comparison to a closed shape space, where all possible variations are known, these fields are far more comprehensive because the whole extent of variability of these elements cannot be captured entirely. An example for this is demonstrated in the domains of windows in Section 5.2.

1.4.2 Reconstruction of Architecture

Sometimes it is necessary to understand a single high-quality example in its entirety. The task is to describe a complex model efficiently, namely with the minimal amount of operations necessary. However, as stated before, it is not possible to find the shortest procedural description, so at least an improvement to existing methods should be found to describe models more efficiently.

City modeling is a very generic task. The state of the art for describing cities procedurally are split grammars (see Section 2.3.4). Through split grammars entire cities can be generated by a single click. High-quality examples in this context are monumental buildings. To describe these monuments more efficiently and to increase the expressiveness of split grammars in general, the state of the art of split grammars needs to be improved.

Split grammars are based on applying split and replacement operations, which are most commonly applied to box-formed shapes. Boxes are very simple shapes and are easy to deal with. However, there are many architectural parts, which are very hard to express with boxes and splits only in major axis directions alone. As buildings blocks are inherently convex shapes it is a obvious thing to consider splitting shapes in arbitrary directions. This leads to the use of the generalization of boxes, namely convex polyhedra, as basic shapes for split grammars. This extension to split grammars is discussed in Section 4.2.

Additionally, curved and deformed buildings are very hard to describe in a short efficient way through split grammars. In curved building designs, repetitions of elements follow the course of the building and to achieve this, a lot of manual fine-tuning is necessary. However, this fine-tuning conflicts with the desire to describe the building with the minimal amount of operations possible. In this thesis I will present an extension to the state of the art that incorporates free-form deformations in the split grammar formalism in Section 4.3.

In the context of this thesis several high-quality building examples have been analyzed, these include, for example, the Louvre (see Section 5.1 and 6.2.1) and the famous Rialto bridge in Venice (see Section 6.2.2). Furthermore, a simplified version of the Great Wall of China (see Section 6.2.3) has been realized to highlight the strengths and advantages of the inclusion of free-form deformations into the split grammar formalism. A procedural description through split grammars of regular buildings and building parts is presented in Section 6.3.

1.4.3 Interactive Procedural Modeling

Many designers and artists are experts in designing complex shapes and can therefore profit from procedural technologies. There are many different ways to obtain a formal description of such shapes, but procedural modeling is tightly connected to programming and programming is rarely an expertise of artists and designers. These non-expert users need a way to convey their implicit knowledge to other people by transferring it into explicit knowledge. Procedural models are an explicit description of procedures and interrelations and are therefore an ideal way for expressing complicated implicit knowledge. To allow these users the creation of procedural models, an intuitive modeling interface is needed that creates code automatically in the background and supports navigation without the need to display complex graphs that represent the scene. For modeling to be most intuitive, interactions should be based on direct manipulations on the explicit three-dimensional model only.

In the Ambient Assisted Living project called V2me the need arose to create animated environments for virtual coaching applications interactively. In course of this project, I extended the GML Compositor, a tool that supports direct manipulations and incorporates several modeling vocabularies, to create scene graphs procedurally in an interactive way (see Section 6.5). Procedural scene graphs have been, furthermore, utilized in this system to create interactive animated exhibits for museums in a Cultural Heritage context (see Section 6.4). Within this system, split grammar inspired modeling operations have been successfully applied to reconstruct façades of the Louvre (see Section 5.1 and 6.2.1). The foundations and capabilities of the GML Compositor are explained in Section 4.4.

Especially, the two applications of procedural scene graphs show that procedural techniques can be applied successfully in domains that are not necessarily tightly connected to computer graphics. This way, non-expert user groups are enabled to create procedural models.

1.5 Structure of this Document

Before providing solutions for the variety of problems presented in this introductory chapter important procedural approaches in visual computing are reviewed in Chapter 2. Afterwards, technical building blocks that are necessary for the realization of the provided solutions are provided in Chapter 3. As foundation for the solutions provided, procedural techniques that were utilized in realizing the tasks are presented in Chapter 4. Chapter 5 provides a case study on how to formalize a new shape domain by presenting a generic method to do so. Finally, the solutions for the examined problems will be presented in Chapter 6. This thesis will conclude in Chapter 7, which discusses the limits and potentials of procedural modeling techniques and gives an outlook on further work to pursue the long-term goal of visual computing.

Synopsis

We have identified procedural modeling as the most promising way to obtain an editable digital representation of the world as well as of the magnitude of things we can imagine. In comparison to other methods, procedural modeling is superior in the domain of man-made shapes because a concise and semantic description can be provided. Through this high-level representation only few degrees of freedom are required to describe the essence of a shape. These advantages, however, come at a price. The main drawback of the procedural approach is the conceptual complexity of the underlying task, namely understanding shapes. A big initial planning effort is necessary to describe a shape. This effort, however, becomes profitable as soon as variations of this analyzed shape are required.

In this thesis problems that belong to three representative problem areas are analyzed to pursue the goal of reaching an editable digital representation of world. These three areas are the understanding of shape spaces, the reconstruction of architecture, and interactive procedural modeling. The main contributions for these three areas are the formalization of a method to acquire procedural building blocks from a set of shape exemplars, extensions to the state of the art of split grammars, and a method to describe procedural animated scenes in an interactive way.

Chapter 2
Previous and Related Work

Contents

Abstract. Procedural techniques are more and more used in all kinds of applications. This section provides an overview on what has been done in relation to the term “procedural modeling”. Despite the vast field of applications in the field of visual computing, the whole potential of procedural modeling has not been exhausted so far. The sum of works done, however, shows the usefulness of this approach. In visual computing, the most common utilization of procedural techniques is creating three-dimensional shapes by describing the modeling process itself. Aside from modeling, procedural techniques can furthermore be used for example for shaders, textures, simulations, particle systems and animations. Procedural techniques are not just limited to the domain of visual computing, however, usage in other domains is very fragmentary, but the trend is increasing continuously. Even though no common theory exists so far, this development indicates a deep change that is happening in the way mankind is embracing new domains. It is infeasible to start from zero, so this section will focus on the use of procedural techniques in visual computing with major interest in fundamentals of procedural modeling by reviewing popular techniques and systems.

2.1 Fields of Application

It is important to distinguish between the terms *parametric modeling*, *procedural modeling* and *generative modeling*.

Definition 2.1 ***Parametric modeling*** *describes the process of creating shapes by combining specific shapes, from which each has a set of well-defined parameters, each with its own range of valid values.*

Definition 2.2 ***Procedural modeling*** *describes shapes through coupling procedures with well-defined in- and outputs together. These procedures are executed in a clearly defined order and have a set of parameters that influence the outcome.*

Definition 2.3 *In* **generative modeling** *shapes are generated through a sequence of geometry-generating modeling operations, called* **operators**. *Shapes are specified through the construction process, which consist of all applied modeling operations.*

In terms of generality, parametric modeling is a subset of procedural modeling, which is again a subset of generative modeling. In parametric modeling shapes are parametric, but object creation is done manually. Parametric shapes can be combined to form complex parametrized shapes. A central term in this regard is the concept of the so-called *feature*. A feature is represented by geometry with attached parameters and is influenced by parameter changes.

Within procedural modeling the creation of parametric shapes can be automated. This is done by defining the parameter flow through a script or a special type of graph. Procedural modeling is therefore equivalent to programming itself. Through this approach it is possible to introduce structural changes to the models, which is not possible with parametric modeling.

Finally, the generative approach is the most universal. Generative modeling introduces a paradigm shift by using geometry-generating operations instead of object primitives. Through this, the creation of procedural shapes can be automated by automatic code generation. This way it is possible to describe entire editors with different modeling tool sets for procedural models, where the whole construction process saved instead of the final three-dimensional model. It is important that the set of provided operators is closed, which means that further modeling operations can be applied to any valid shape, resulting again in a valid shape. Snyder and Kajiya [SK92] provided a generative modeling interface for three-dimensional curves, surfaces, and solids, which inspired Havemann [Hav05] to pursue the same principle for meshes.

An example to illustrate the difference is the following. Let us imagine a shape that is designed to have drill holes. In a parametric modeling context the positions of a pre-defined number of holes can be changed through parameters. In a procedural modeling context the amount and position of the drill holes can be arbitrary and is not limited as within a parametric context. In the universal generative modeling context, for example, a drill hole shape editor for these models can be described and used through the same underlying scripting language.

Use Case for Parametric Modeling: Computer-Aided Design. Parametric modeling techniques are already in use in a lot of commercial and non-commercial products. The most important category of such products is *computer-aided design (CAD)* software. Computer-aided design software is used in a huge variety of fields such as architecture, engineering and construction. The main goal of using computer-aided design software is *parametric product development*. Products are generated through parametrized two-dimensional vector plans or parametrized three-dimensional solids and surfaces, which encapsulates the core idea of parametric modeling. The amount of computer-aided design software on the market is huge and there are specialized versions for every domain. The most prominent pieces of software are probably AutoCAD [Aut13b] from Autodesk and CATIA [DS13] from the French developer Dassault Systemes.

General and Domain Specific Modeling Tools. There are also general three-dimensional modeling tools that feature procedural techniques to some extent like the commercial Maya [Aut13c] and 3D Studio Max [Aut13a] from Autodesk. Both, 3D Studio Max and Maya, feature their own internal scripting language to enable procedural modeling. Maya, in contrast to 3D Studio Max, is completely built upon its scripting language and is therefore more suited to create objects procedurally. The free software Google Sketchup [Goo13c] is an easy to use modeling tool set with intuitive controls. It discards all the information necessary for procedural modeling during the modeling process because it simply has not been designed for procedural creation of objects. In contrast to the general modeling tools, domain specific procedural modeling environments are also present. Most notably here are Esri CityEngine [Esr13] in the modeling domain of cities and buildings and Xfrog [Xfr13] for modeling of organic objects like trees and plants.

Procedural Techniques in Computer Games. Procedural techniques are increasingly used in computer games and movies. This is due to the high database amplification, the ease with which whole families of objects can be described and other advantageous features like editability and re-useability. In particular, video games use procedural techniques, but not only for three-dimensional content generation. Procedural textures and procedural animations are just but two of different fields of application. Games that are worth mentioning for utilizing procedural animations are Spore and the Assassin's Creed series. In Spore the users can create their own creature species in an editor. Based on the amount and position of the extremities each creature has its own set of unique animations. In Assassin's Creed the walking, running and climbing animations of the characters adapt to different terrains of the game world.

Also worth mentioning here is the game .kkrieger [.th13a] from the German developer .theprodukkt. This game makes extensive use of procedural techniques and was developed using their tool called .werkkzeug [.th13b]. All content – from textures over sound effects and music to three-dimensional models – is created procedurally. The entire game uses only 97.280 bytes of disk space in contrast to other modern games that take up several gigabyte of storage. This difference is, of course, biased because the graphics and model quality is different, but it remains a fact that through describing content procedurally a lot of storage space is saved. There is, however, one major drawback of this approach: the extensive loading time. All content has to be generated before use. Therefore, a trade-off between time and space is necessary. A screenshot of this game is shown in Figure 2.1. Another game that relies heavily on procedural generation of environments is the recently announced game No Man's Sky by Hello Games [MDDR13]. In their game planets, together with all lifeforms on it, are created procedurally and are therefore unique.

2.2 Scripting Languages

Three-dimensional modeling software packages like Autodesk Maya [Aut13c], or Google Sketchup [Goo13c] provide a set of modeling tools to ease the modeling process. Complicated and repetitive tasks are, however, not always suited to a three-dimensional graphical model-

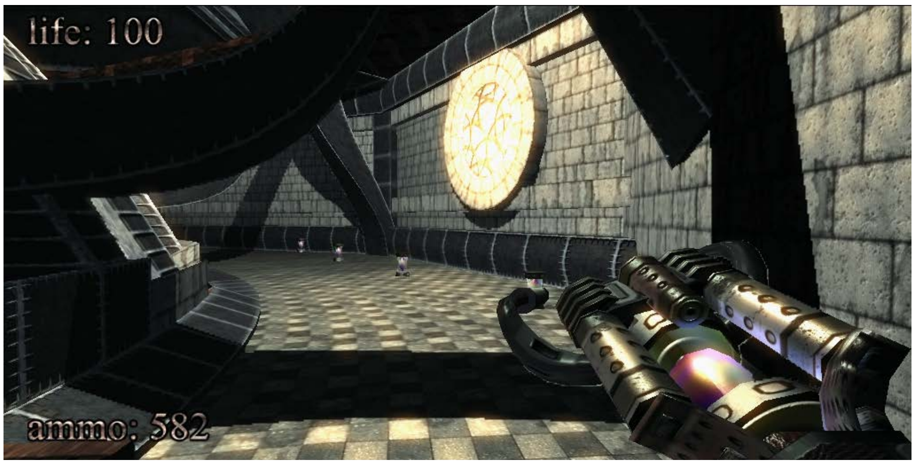


Fig. 2.1 Screenshot from the fully procedural game .kkrieger [.th13a]. The graphics and details are not up to par with state of the art video games, but at its release date in 2004 the procedural generation of all in-game content was quite an achievement.

ing interface. Therefore scripting languages are utilized in procedural modeling environments. Some systems use common scripting languages like Python or Ruby, but the need of different features introduced several new scripting languages or scripting language dialects. Such scripting languages include for example *OpenSCAD* [Mar13] for creating three-dimensional computer-aided design models procedurally, *Processing* [Ben13], a scripting language for visual artists, or domain specific languages like *CGA Shape* [MWH*06] or G^2 [KPK10], which are used in the domain of split grammars. Processing, for example, started out with the goal to teach students programming, but due to its comprehensive library of visual elements it turned into a tool used by visual artists. Processing is used to realize a great number of interactive art installations today. The limits of processing lie within mesh modeling because the scripting of mesh operations tends to become difficult to handle.

The scripting language used for procedural modeling throughout this thesis is the *Generative Modeling Language (GML)* by Havemann [Hav05]. The GML is inspired by Adobe PostScript but with the focus on describing three-dimensional shapes. The GML was specifically designed for generative modeling and features automatic code generation to avoid the problem of manual programming. This makes this scripting language ideal for the tasks of procedural and generative modeling. The GML will be explained in detail in Section 4.1.

The Persistent Naming Problem. Within three-dimensional modeling it is always necessary to refer to an element out of a set of objects. If this is done in a static way – on an index-basis – this reference is valid as long as the size of the set remains the same. However, procedural modeling describes many different variations of a model and therefore deals with highly dynamic sets of elements. An index-based reference introduces the so-called *persistent naming problem*.

Persistent naming problems are also rooted within history-based modeling systems. Commercial modelers like Maya [Aut13c] and AutoCAD [Aut13b] from Autodesk feature their own embedded languages, namely the *Maya embedded language (MEL)* for Maya and *AutoLisp* for AutoCAD, to add specialized modeling functionality through scripting. Through these languages modeling can be sped up, but the main problem lies within the combination of modeling done by scripting or changing parameters and manual modeling.

The problem is that geometric and topological features, like displaced vertices, need to be characterized and identified again when the model is re-evaluated with new parameters. Referencing such entities on a per-index basis creates the aforementioned problem. Primitives in Maya, for example, have their own parameters, like the degree of tessellation. If a vertex is displaced by manual modeling and the tessellation is changed afterwards, then this displacement cannot be retained at that position. This is due to the fact that the displacement is saved in correspondence to a vertex index and through the re-tessellation the indices of the vertices are changed. To avoid these problems the identity of geometric and topological features needs to be retained by other means like ray casting.

The persistent naming problems associated with features in parametric modeling are discussed in chapter 21 of the *"Handbook of Computer Aided Geometric Design"* [HJA02]. Additionally, Marcheix and Pierra provide a survey on of existing approaches on persistent naming in parametric systems [MP02].

Alternatives for Scripting Procedural Models. Scripting is not the ideal way to approach procedural modeling for everyone. Artists may not be fond of learning how to use scripting languages to realize their ambitions. This led to procedural modeling environments with the goal to reduce the coding effort to a minimum. The modeling packages Houdini [Sid13] and Grasshopper [Rob13a] (an extension to the modeling tools of Rhinoceros [Rob13b]) utilize graphs as an extension to clarify and create the model's hierarchy; scripting is not necessary. In this data flow graph, data is passed from node to node and so parameters of different entities are filled with values. By evaluating these graphs, the necessary scripting code is generated automatically. These graphs in these *visual programming languages*, however, tend to become very huge and confusing for complicated models. Therefore, they are not necessarily easier to read than textual programs as stated by Green and Petre [GP92].

2.3 Grammar-Based Procedural Modeling Approaches

An obvious way for procedural top-down modeling is a hierarchical replacement scheme. Coarse objects, which approximate the final shape, are replaced by finer and more detailed objects until the desired level of detail is reached. Computer science features a very powerful tool to achieve such replacement schemes: grammars.

Within grammars, symbols (or shapes) that are identified by a label are replaced by other entities that again carry labels. By executing these replacement steps, a sequence of symbols is reached starting from a single start symbol. This conceptual simplicity is the biggest advantage of grammar-based systems. There exist several trade-offs between expressiveness and limitation to determine the power of different grammar types. These reach from very restricted types that are not Turing complete to types that are as powerful as programming itself.

In procedural modeling, grammars are a useful tool to describe structures and configurations of complex shapes. Especially hierarchical structures like façades are easily expressed by grammars. This section first gives an introduction to formal grammars (see Section 2.3.1) and then explores applications of grammars in the field of procedural modeling, among which are Lindenmayer systems (see Section 2.3.2), shape grammars (see Section 2.3.3) and graph grammars (see Section 2.3.5).

2.3.1 Formal Grammars

In formal language theory a formal grammar, as introduced by Chomsky in 1956 [Cho56], is a set of rules that generates a string of symbols.

Definition 2.4 *A* ***formal grammar*** *G is a tuple (N, Σ, P, S) where:*

- *N denotes a finite set of* ***non-terminal*** *symbols,*
- *Σ denotes a finite set of* ***terminal*** *symbols (the* ***alphabet*** *or* ***vocabulary****),*
- *P denotes a finite set of production (or replacement) rules $\alpha \rightarrow \beta$ where each side consists out of these symbols, and*
- *$S \in N$ is a distinguished non-terminal symbol, which acts as the* ***start symbol****.*

A production rule $\alpha \rightarrow \beta$ can be applied by replacing an occurrence of the symbols on the left-hand side (α) with the symbols on the right-hand side (β). A sequence of applications of replacements rules is called a *derivation*. A sequence of terminal symbols that is created by a derivation from the starting symbol is called a *word* or a *string*. The empty word is usually represented by an ε.

A *non-terminal* symbol, which is usually represented by a capital letter, in a produced string indicates that a production rule can still be applied. A *terminal* symbol, which is commonly represented by a lower case letter, on the other hand, means that no production rule can be applied to this symbol alone. When there are no non-terminal symbols present in the produced string, no rule can be applied any more.

A formal grammar, furthermore, defines a formal language.

Definition 2.5 *The* ***formal language*** *$L(G)$ of a grammar $G = (N, \Sigma, P, S)$ is the (usually infinite) set of all words that can be derived in finite amount of steps from the start symbol S by application of the rules from the set P.*

Chomsky classified formal grammars into types known as the *Chomsky hierarchy* [Cho56]. Each level of this hierarchy includes grammars with increasingly strict production rules, which results in fewer formal languages that can be expressed. The following levels form this hierarchy:

- *unrestricted grammars* (Type 0) can generate any language that is capable of being expressed by a Turing machine. There are no restrictions for the rules of the form $\alpha \rightarrow \beta$, except that $\alpha \neq \varepsilon$.
- *context-sensitive grammars* (Type 1) contain rules of the form $\alpha A \beta \rightarrow \alpha \gamma \beta$ with $A \in N$ and $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$ (string of terminal and non-terminal symbols), where for a set A, the *Keen closure* A^* is the set which contains all strings of elements from A and additionally the empty word ε. The string γ must not be empty.
- *context-free grammars* (Type 2) contain rules of the form $A \rightarrow \gamma$, with $A \in N$ and $\gamma \in (N \cup \Sigma)^*$.
- *regular grammars* (Type 3) are the most restrictive grammars in the hierarchy. It only accepts rules of the form $A \rightarrow \varepsilon$, $A \rightarrow a$ and $A \rightarrow aB$ with $A, B \in N$ and $a \in \Sigma$.

Of particular interest in procedural modeling are regular, context-free, and context-sensitive grammars. Figure 2.2 shows examples for rule sets of these grammar types, in which non-terminal symbols are written in upper case and terminal symbols in lower case. Figure 2.2(a) shows a regular grammar, Figure 2.2(b) a context-free grammar, and Figure 2.2(c) a context-sensitive grammar. The respective languages are $\{a^n | n \geq 1\}$, $\{a^n b^n | n \geq 1\}$, and $\{a^n b^n c^n | n \geq 1\}$. The starting symbol for these grammars is always the non-terminal symbol S. Even though the languages are quite similar, producing exact the same amount of different symbols is a constraint that affects the complexity of the corresponding grammar.

$$\begin{array}{c}S \rightarrow aS\\ S \rightarrow a\\ \\ L = \{a, aa, aaa, \dots\}\\ \text{(a)}\end{array}\qquad \begin{array}{c}S \rightarrow aSb\\ S \rightarrow ab\\ \\ L = \{ab, aabb, aaabbb, \dots\}\\ \text{(b)}\end{array}\qquad \begin{array}{c}\begin{array}{lll}S \rightarrow aSBC & aB \rightarrow ab & \\ S \rightarrow aBC & bB \rightarrow bb & cC \rightarrow cc\\ CB \rightarrow BC & bC \rightarrow bc & \end{array}\\ \\ L = \{abc, aabbcc, aaabbbccc, \dots\}\\ \text{(c)}\end{array}$$

Fig. 2.2 Examples for rule sets of regular (a), context-free (b), and context-sensitive grammar (c). Non-terminal symbols are written in upper case and terminal symbols in lower case. The starting symbols is always the non-terminal symbol S. The respective languages are $\{a^n | n \geq 1\}$ in (a), $\{a^n b^n | n \geq 1\}$ in (b), $\{a^n b^n c^n | n \geq 1\}$ in (c). Constraints, like producing the same amount of symbols throughout words, affects the complexity of the grammar.

2.3.2 Lindenmayer Systems

An Lindenmayer system (or L-system) is a parallel rewriting system named after their inventor Lindenmayer [Lin68]. An L-system is a variant of a formal grammar and is very similar to a *semi-Thue grammar*. They are most famously used to model growth processes of plants, but can also be used to describe self-similar fractal forms [Pru86], which are also very common in nature [Man82].

Definition 2.6 *A parametric* ***L-system*** *is a tuple* (V, ω, P)*, where*

- *V is a set containing all symbols that can be replaced (the* ***alphabet****),*
- *ω is a string of symbols of V that marks the initial state of the system (the* ***axiom****), and*
- *P is a set of production rules $\alpha \rightarrow \beta$, $\alpha, \beta \in V^*$. Those rules are used to replace a* ***predecessor*** *(α) with a* ***successor*** *(β).*

For elements $A \in V$ for which no production rule exists, an identity rule $A \rightarrow A$ is assumed. Symbols with such identity rules are called ***constants*** *(or* ***terminals****).*

The main difference to the other grammars mentioned so far is, that at each iteration as many rules as possible are applied simultaneously. The set of possible derivations from the starting symbol is a strict subset of the language that would be generated if each rule would be applied once at a time. This is the most distinguishing feature between a formal language and an L-system.

L-systems are very powerful tools in the domain of plant modeling because they are very close to biological simulations. The produced results are, furthermore, very plausible. However, it is not easy to predict the outcome and therefore the final shape of the plant. Moreover, L-systems are very complex and there is only little control over the rewriting process.

Variations of L-Systems. There are many variations and extensions to the basic L-system formalism. An L-system is called an *context-free L-system* when the predecessor consists out of one single element. *Deterministic L-systems* contain for each symbol in the alphabet exactly one production rule and consequently always reaches the same result when starting from the same axiom. L-systems are called *indeterministic* or *stochastic* otherwise. Rules have to be chosen randomly based on probabilities for each rule application. In *context-sensitive L-systems* rules are not only chosen based on the predecessor, but also of the context in which the predecessor resides. *Parametric L-systems* add a condition to each rule, which needs to evaluate to $true$ for the rule to be applied. These systems are more expressive and are commonly used.

Visualization of L-Systems. L-systems are often visualized using *turtle graphics*, which is a method for generating vector graphics. The cursor (or the turtle) has three attributes: a location, an orientation and a "pen", which itself carries attributes like color and width. The turtle executes commands like "move one unit forward", or "turn right 90 degrees" always drawing a line along the path it moved. To increase the expressiveness, stacks of the turtle's attributes can be used. A stack of turtle positions, for example, can be used to simulate more turtle paths at once.

Consider an L-system (V, ω, P), with $V = \{0, 1, +, [,]\}$, $\omega = 0$ and the following rules in the set P:

$$\begin{aligned} 1 &\to 11 \\ + &\to ++ \\ 0 &\to +1[0]0 \end{aligned}$$

Starting with the axiom ω all applicable production rules are used simultaneously to generate the next step. Starting with the axiom $\omega = 0$, the following strings are derived for for the first three recursions:

$$\begin{aligned} \text{axiom} &: 0 \\ \text{step 1} &: +1[0]0 \\ \text{step 2} &: ++11[+1[0]0]+1[0]0 \\ \text{step 3} &: ++++1111[++11[+1[0]0]+1[0]0]++11[+1[0]0]+1[0]0 \end{aligned}$$

To visualize these strings, each symbols needs to be interpreted by a turtle graphic. For this example, one set of instructions for the turtle can be:

- 0: draw a green line segment of length one unit
- 1: draw a line segment of length two units
- +: increase the width of the pen
- [: push a new turtle position, rotate the current position by 45 degrees to the left and reset the pen width
-]: pop the current turtle position, rotate the position by 45 degrees to the right and reset the pen width

The first four iterations visualization of this L-system with the presented instruction set is illustrated in Figure 2.3(a).

Applications of L-Systems for Growth Processes and Plants. In the field of automatic plant and tree generation two important persons pioneered by using L-systems: Lindenmayer and Prusinkiewicz [Lin68, PL90]. Plants can be efficiently represented by L-systems and thus leading to a high database amplification. L-systems were extended by several authors by different formal grammar aspects like parameters, stochastic rule application, and context-sensitivity [Smi84, PHHM97]. Furthermore, environmental effects such as pruning or insect attacks can be considered as well (as in Prusinkiewicz et al. [PJM94]). Environmental factors like the com-

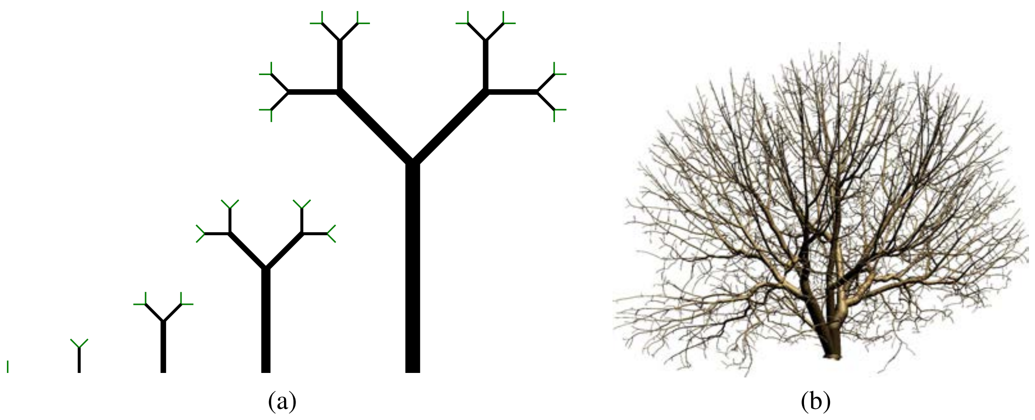


Fig. 2.3 Visualization through turtle graphics of generations 0 to 4 of an L-system that describes simple plant-like structures (a). In comparison a more complex three-dimensional model (b) can be realized through the technique of Runions et al. [RLP07].

petition for light and space above the surface and competition for water between the roots in the soil were introduced to L-systems by Měch and Prusinkiewicz in 1996 [MP96]. This new kind of L-system incorporating all these extensions is called *open L-system* and is the foundation of state of the art plant modeling. Open L-systems led to the introduction of the plant modeling language *cpfg* [PHM00, PKMH00]. In 2001 it was furthermore extended to also take positional information like posture or silhouettes into account [PMKL01].

Runions et al. [RLP07] pursued the competition for resources further by using space colonization algorithms that determined the growth by distributing resources in the environment (see Figure 2.3(b)). Bornhofen et al. [BL09] utilize L-systems to define a plant model targeting evolutionary experiments involving environmental constraints. L-systems are also applied by Krecklau et al. in their introduction to their scripting language G^2 [KPK10]. They use a turtle that applies free-form deformations to generalized cylinders to model plants.

Further Applications of L-Systems. With modeling of growth processes of plants being the main application of L-systems there are some other approaches worth mentioning. In 2001 L-systems were used by Parish and Müller [PM01] to procedurally model cities with the main focus on the underlying street network. Interpreting a street layout as a mathematical tree, which is an implication of using a hierarchical replacement scheme such as L-systems, however, is not necessarily the correct representation as Alexander pointed out in his article *"A City is not a Tree"* [Ale65]. He states that for a representation of cities in their entirety – including all systems among which are water pipes, postal services, and society – the generalization of a tree, namely a *semi-lattice*, is more appropriate.

Applying L-system rules to meshes leads to *mesh-based parametric L-Systems* in the context of *procedural mesh growing* [Mai02, TMW02, MDH*10]. They are an extension to parametric L-systems that replace connected symbols (faces of a mesh). With this mechanism highly tessellated complex geometry can be inserted into meshes where fine detail is needed. Menz et al. [MDH*10] also presents an interactive graphical user interface in this field.

2.3.3 Shape Grammars

A shape grammar is very similar to a standard context-free (in some cases a context-sensitive) formal grammar. However, instead of being defined over an alphabet of symbols and generating one-dimensional strings of symbols, shape grammars are defined over an alphabet of shapes of arbitrary dimension creating other shapes of any dimension. But what exactly are these shapes in the context of shape grammars? The understanding of the term "shape" changed over time. At first shapes were defined just as a set of lines, but they evolved into complex structures carrying various labels, attributes, and forms of geometry in state of the art shape grammars.

Classical Shape Grammars. The idea of a shape grammar originated in the paper *"Shape Grammars and the Generative Specification of Painting and Sculpture"* by Stiny and Gips in 1972 [SG72]. This concept was formulated later by Stiny [Sti77, Sti80] leading to definitions of shapes and shape grammars respectively.

The definition of a shape context of a classical shape grammar is the following.

Definition 2.7 *A* ***shape*** *(in a classical shape grammar) is a limited arrangement of straight lines defined in a Cartesian coordinate system with real axes and an associated euclidean metric.*

This simplistic shape representation is realized through a set of non-collinear lines in Euclidean space. For rule application in shape grammars it is very important to detect a given shape as a sub-shape in another shape and replace it. This shape representation helps to ease the definition of identity and sub-shape relations. Boolean operations (union (+), intersection ($\cap$) and difference (-)) between two shapes are also a necessary tool for replacing parts of shapes. Shapes can also be transformed using compositions of affine transformations.

Labels on points are used to realize features of non-terminals within a shape. A *labeled point* $p = \{v, A\}$ is a point v with a symbol A associated to it. Two labeled points are the same if, and only if, the two points and the two symbols coincide. A *labeled shape* σ, on the other hand, is a pair $\langle s, P\rangle$ of a shape s and a set of labeled points P. Two labeled shapes are the same if, and only if, both shapes and sets of labeled points are the same. A labeled shape $\langle s, \emptyset\rangle$ has no labeled points nor symbols attached to it and is therefore a terminal shape. For two labeled shapes σ_1 and σ_2 Boolean operations, identity, and sub-shape relations always take the shapes s_1 and s_2 as well as the sets of labeled points P_1 and P_2 into account.

The definition of the classical shape grammar follows the definition of a context-free grammar:

Definition 2.8 *A* ***(classical) shape grammar*** *SG is a tuple* (S, L, R, I)*, where*

- *S is a finite set of shape elements,*
- *L is a finite set of symbols,*
- *R is a finite set of* ***shape rules****, and*
- *I is the* ***initial shape*** *consisting out of elements from* $S^* \cup L$*.*

The set S^* is formed by finite arrangements of elements $t_i \in S$, for $0 < i \leq |S|$. Any element t_i can be contained multiple times in any arrangement with different position, orientation, and scale. Elements of S^* are called *terminal shape* elements, therefore S^* is the vocabulary of a shape grammar.

The set R of shape rules defines rules $\alpha \to \beta$, in which a labeled shape α gets replaced by another labeled shape β. A shape rule can be applied to a labeled shape σ when the shape on the left-hand side (α) of the rule is geometrically similar to a part of σ. The rigid transformation that transforms the shape α to match the similar occurrence of α in σ, needs to be applied to the shape of the right-hand side (β) before substituting the similar occurrence of α for β in σ. In short, for a rule $\alpha \rightarrow \beta$, if a similarity transformation φ exists such that $\varphi(\alpha)$ occurs in the shape σ currently processed, the rule can be applied by replacing $\varphi(\alpha)$ by $\varphi(\beta)$ yielding the new shape.

Finally, the *language* $L(SG)$ of a shape grammar $SG = (S, L, R, I)$ is the potentially infinite set of all shapes containing no symbols from L, that can be derived from I. The analogy to formal grammars is trivial.

For understanding classical shape grammars better, an example is provided in Figure 2.4. Consider a shape grammar $SG = (S, L, R, I)$ with $S = \{\square\}$, $L = \{\bullet\}$, R as in Figure 2.4(a), and I as in Figure 2.4(b). Some shapes in $L(SG)$ are shown in Figure 2.4(c). A derivation of one of these shapes is illustrated in Figure 2.4(d). The identifier of the applied shape rules is shown above the arrows.

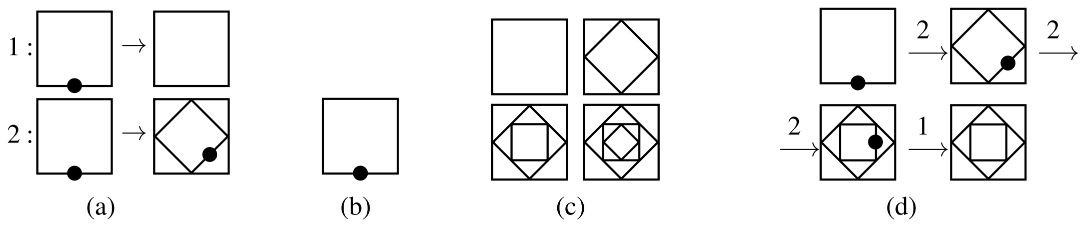


Fig. 2.4 A simple shape grammar that inscribes squares into squares (inspired by the guiding example of [Sti80]). For a shape grammar $SG = (S, L, R, I)$ with $S = \{\square\}$ and $L = \{\bullet\}$, (a) describes the shape rules R and (b) describes the initial shape I. Some shapes of the language $L(SG)$ are shown in (c). A derivation of one terminal shape is illustrated in (d) with following rule application sequence: 2-2-1.

Parametric Shape Grammars. Stiny, furthermore, introduced parametric shape grammars [Sti77, Sti80], which are the foundation of the shape grammars most used today. By parametrizing labeled shapes, a whole family of shapes can be expressed. Such a *parametrized labeled shape* is obtained by allowing the coordinates of labeled points and lines within shapes to be variables. A member of such a family can be determined by a variable assignment g of real values.

Definition 2.9 *A* ***parametric shape grammar*** *$PG = (S, L, RS, I)$ is a shape grammar $G = (S, L, R, I)$ (see Definition 2.8) where the set of shape rules R is replaced by a set of rule defining* ***rule schemata*** *RS.*

A shape rule schema can be understood as a family of specific rules replacing parametrized labeled shapes. A rule schema $\alpha \rightarrow \beta$ for parametrized labeled shapes α and β generates a specific rule $g(\alpha) \rightarrow g(\beta)$ for a variable assignment g that yields specific labeled shapes $g(\alpha)$ and $g(\beta)$. This generated shape rule can be used in the usual way.

The formalism of parametric shape grammars (see Definition 2.9) is still valid for state of the art examples of shape grammars. The definitions of parametric labeled shapes and shape rule schemata are the ones that undertook several changes over the years.

Shape Grammar Examples. Shape grammars have since been used for describing various design spaces such as chairs [Kni80], coffee machines [AC98], floral ornaments [WZS98], plant ecosystems [DHL*98], and Harley-Davidson motorcycles [PC02].

Shape grammars, furthermore, found their application in the field of architectural analysis. The Palladian grammar from Stiny et al. [SM78] provides the rules to recreate the ground plans of Palladio's villas. Further shape grammars on prominent architecture are the recreation of Frank Lloyd Wright's prairie houses by Koning et al. [KE81], the bungalows of Buffalo by Downing et al. [DF81], Queen Anne houses by Flemming [Fle87], Christopher Wren's city churches by Buelinckx [Bue93], and Alvaro Siza's houses at Malagueira by Duarte [Dua02, Dua05]. Duarte et al. [DRS07] also used a shape grammar to describe the structure of the city Medina of Marrakech.

2.3.4 Split Grammars

In case of computer implementation of shape grammars, the before mentioned sub-shape matching procedures are difficult to implement. A more amenable approach to computer implementation are *set grammars* [Sti82], in which shapes are treated as symbolic objects that can be replaced by matching their label only. Shapes are no longer just the (parametrized) geometry with an assigned label, but more a structure of different elements representing a *non-terminal shape*.

Definition 2.10 *A* ***scope*** *is a reference frame that is part of a non-terminal shape description. Shape rules take the scope into account when calculating measurements for their application.*

Definition 2.11 *A* ***non-terminal shape*** *is a tuple (L, S, C, G, A) consisting of*

- *an arbitrary label L,*
- *a shape S describing the scope,*
- *a local coordinate system C,*
- *a geometry G, and*
- *a set A of attributes (name-value pairs).*

In the formulation of a non-terminal shape the separation of scope and geometry does not need to be explicit. Geometry can also be created from the shape of the scope when the non-terminal shape is finally replaced by a terminal shape.

Shape grammars are well-suited for describing hierarchical structures such as many forms of architecture or particularly buildings. This leads to a coarse-to-fine procedural modeling approach using split operations on shapes with, for example, their (axis-aligned) bounding box as scope. Using box-shaped scopes for modeling architecture is quite obvious because most architectural elements can be encapsulated in a building block that is shaped like a box. By using axis-aligned boxes as scope the local coordinate system is defined implicitly through the three major-axes directions defined by the scope. In this thesis this concept will be referred to as a *split grammar*, which have been introduced by Wonka et al. [WWSR03].

Definition 2.12 *A* ***split*** *operation is a decomposition of a shape a into several other disjunct shapes $a_1, \ldots a_n, n \geq 1$. The arrangement of those new shapes fills the same volume as the shape a did.*

Definition 2.13 *A **split grammar** is a shape grammar that supports two different kind of rules:*

- *A **split rule**, i.e. a rule $\alpha \rightarrow \beta$ where α is a non-terminal shape, and β contains one or more shapes which are the result of a split operation applied to α. For these shapes the containment property has to be fulfilled (the union of the shapes representing β fills the same volume α did).*
- *A **conversion rule** is a rule $\alpha \rightarrow \beta$ that substitutes one shape for another, i.e. α is a non-terminal shape, and β is a different shape as α. That shape in β has to be contained in the volume of the replaced shape in α.*

The limitations introduced by box-based split grammars and a transition to a more general class of objects, the convex polyhedra, are discussed in Section 4.2.

Split Operations for Boxes. Specific split operations are *subdivide* and *repeat* (see Figure 2.5) that operate on the three principal axes of the local coordinate system of the shape they are applied to. The subdivide operation splits a shape along a direction into a fixed number of parts with specified lengths. Lengths can be given in absolute (lengths do not scale) and relative (lengths do scale) values. With relative values it is possible to fill remaining space and to split a shape into specific proportions. The repeat operation, on the other hand, splits a shape along a direction into a variable number of shapes with length of at least the given value. For these operations the so-called *containment property* is fulfilled, meaning that the union of the shapes resulting from the split operation fills the volume that the shape, on which the split operation was used, had.

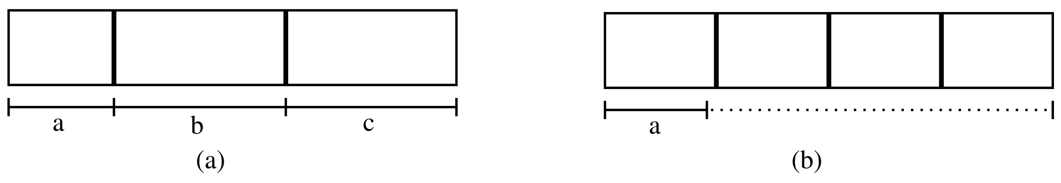


Fig. 2.5 The subdivide operation (a) splits a shape along a direction into a fixed number of parts with specified lengths. By using relative measurements the remaining space can be padded, or a shape can be split into specific proportions. The repeat operation (b), however, splits a shape along a direction into a variable number of shapes with length of at least the given value.

Development of the Concept of Split Grammars

With split grammars it seems that a consensus has emerged on how to apply shape grammars on procedural modeling of buildings. The following papers will provide extensions to this common core concept to gain more expressivity but also to introduce mechanisms to deal with the shortcomings of this original formalism. These shortcomings will be discussed first before the presentation of the developed concepts. The advances of our research group in the domain of split grammars are explained in Section 4.2 and 4.3.

Shortcomings of Split Grammars. The concept of these context-free split grammars, which are mainly used for building generation and reconstruction, shares its limitations with standard grammars, of course, but also introduces new difficulties.

An inherent problem of split grammars is their *context-free nature*. Connections between elements deep down in the hierarchy is only possible with non-context-free means. Dependencies between different levels, predecessors, or branches are hard to resolve using the depth-first evaluation usually done by grammars. This way, not even objects in the same level of the hierarchy can react or address to each other.

The combining of objects that are realized through split grammars often introduces the problem of *overlapping structures*. Structures can occlude other structures and make further refinement steps unnecessary or wrong. Imagine two or more three-dimensional shapes that touch it other and together form a building complex. By partitioning each side of these shapes into parts for windows it will happen that some of these spaces are partially or complete occluded by another shape. Inserting a window at these occluded spaces is certainly wrong. Additionally, through the use of modeling operations, which do not fulfill the containment property, like extrude operations, structures can overlap and occlusions can be generated, which are very hard to detect.

Split grammars, furthermore, introduce the problem of *crossing structures*. Especially, when modeling façades vertical and horizontal intersect with each other, which introduces the question of which to split first. An illustrative problem in the case of façades is the question in what direction the original shape is split first: horizontally into floors or vertically into window axes? The problem even increases when decorative elements like ledges begin to intersect in the absence of a clear rule that resolves how two ledges interact. There can be, for example, one superior structure, where the inferior runs over the dominating ledge, or both ledges can be "equal", which results in combination of both. In special cases, there can even be completely unrelated additional decorations at the intersection point.

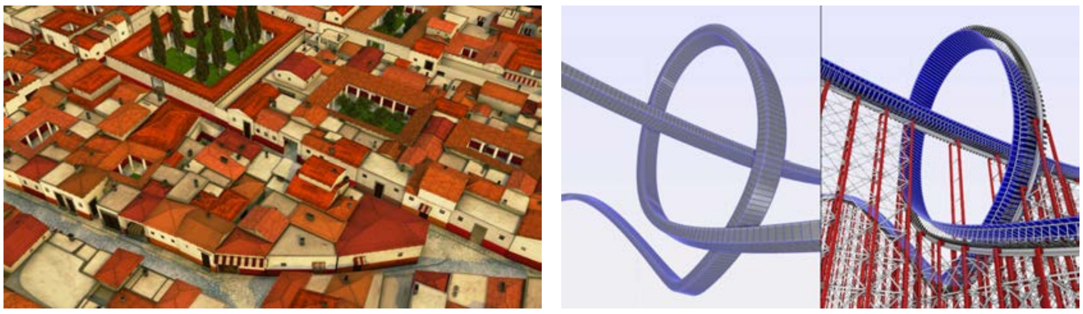
Finally, there is the problem of *exception handling*. Split grammars are ideally to cover repeating structures and elements. However, elements do not necessarily have to be exactly the same. A simple example is the exception of a door in a row of windows in the ground floor, but also in an repetition of recurring design elements, some may introduce slight differences to the otherwise identical shapes.

The Origin of Split Grammars. Wonka et al. [WWSR03] were the first to utilize a split grammar approach to computer generated architecture. For better control of the derivation process and to allow exceptions they applied an attribute system to their grammar. Attributes are either defined at the start symbol, propagated from parent shapes to children shapes, or are set by rules via a special *control grammar*. This separate grammar defines attributes dependent on spacial locators. So it is possible, for example, to give the second floor windows a different style, or to ensure that there is a door at the ground floor. Attributes are used, on the one hand, to carry material information at different granularities and, on the other hand, to steer the derivation process by selecting one specific rule out of a set of matching rules.

The Split Grammar *CGA Shape*. In 2006 Müller et al. [MWH*06] introduced *CGA Shape*, a split grammar for modeling building shells procedurally to obtain large scale city models. This split grammar led to the commercial product Esri CityEngine [Esr13]. This grammar focuses on generating mass models of buildings, which are then further refined using split operations. *Stochastic rules* are applied to create unique buildings based on footprints on a large scale (see Figure 2.6(a)). An important contribution of their work is the *component-split*, their basis for modeling with shapes of different dimension. Three-dimensional shapes, such as loaded meshes, can be partitioned into shapes of lesser dimension, such as faces and edges. The expressiveness of their grammar is enhanced because these lower-dimensional shapes can then be split and extruded for further refinement. To utilize operations like the component-split it is necessary to distinguish between scope and geometry explicitly. They furthermore build upon the before mentioned control grammar from Wonka et al. [WWSR03] by combining it with stochastic replacement rules.

Further important contributions are *occlusion queries* to deal with the problem of overlapping structures, and the *snapline mechanism* to provide a simple solution for context sensitivity. To illustrate the occlusion queries, imagine a building that is composed out of a union of different shapes. Further refinement of a non-terminal shape can be halted dependent of its occlusion level – *"none"*, *"partial"* or *"full"* – to, for example, place windows on a façade. Through the snapline mechanism splitting planes can be snapped to prominent planes in the model to, for example, ensure same floor levels throughout the building. These techniques become possible because replaced non-terminal symbols do not get deleted, but are just rendered invisible to maintain the information on the whole hierarchy of the model.

Müller et al. [MVW*06] furthermore used their approach to reconstruct Puuc buildings that are present in Xkipché. Based on the work of Müller et al., Whiting et al. [WOD09] built a split grammar for structurally-sound masonry buildings, which can be manipulated in physically simulated environments.



(a) (b)

Fig. 2.6 An interpretation of Pompeii (a) reconstructed based on real building footprint by 190 hand-written *CGA Shape* rules by Müller et al. [MWH*06]. The city model is a composition of 36 different terminal objects. The roller coaster (b), which is defined by a spline, demonstrates the ability to handle interconnected structures by Krecklau and Kobbelt [KK11]. To avoid intersection of the pillars with the roller coaster a ray is cast before generating the geometry.

Interconnected Structures in Split Grammars. Later, Krecklau et al. [KPK10] introduced their split grammar G^2. It provides further refinements to the split grammar concept such as supporting different non-terminal shapes to encapsulate different modeling strategies. Besides boxes, they support free-form deformations as a new non-terminal shape class to overcome the limitation of loading meshes that are scaled to the scope they are inserted to. This new class, however, can only be used to deform shapes and to not allow further refinement. To address this problem I will present an extension to the split grammar formalism in Section 4.3.

A solution for problem of missing context sensitivity was presented by Krecklau and Kobbelt in 2011 [KK11]. Their solution for interconnected Structures is based on so-called *containers* – nested arrays – that are passed to the grammar rules. Potential connecting points are appended to these containers and are later processed to generate connecting geometry (see Figure 2.6(b)). These connections are realized with deformable beams and rigid chains featuring several independent parts, whose correct position and orientation is calculated through inverse kinematics.

Further Applications of Split Grammars in the Domain of Architecture. Applications of split grammars are mostly in the domain of architecture. Larive et al. [LG06] use a simple split grammar to describe exteriors of buildings they have created by extruding arbitrary ground plans. Their simplified grammar only has a set of five rules: two repetition rules ("wall grid" and "wall list"), two position rules ("extruded wall" and "bordered wall") and one terminal rule for texturing. Hohmann et al. [HHKF10] use a split grammar to entirely describe specific office buildings (with interiors) at Graz University of Technology. They only use axis-aligned boxes and two different terminal symbols: fill the box with a material or render the box invisible. While generating the model all measurements have been approximated and due to the procedural description of the reconstruction, real measurements can be incorporated later. The result of their reconstruction (see Figure 2.7) has been later used for surveillance applications. Thaller et al. [TKH*11] use split grammars to describe families of buildings through parametric building templates. These templates are then used to optimize the energy footprint for these buildings.

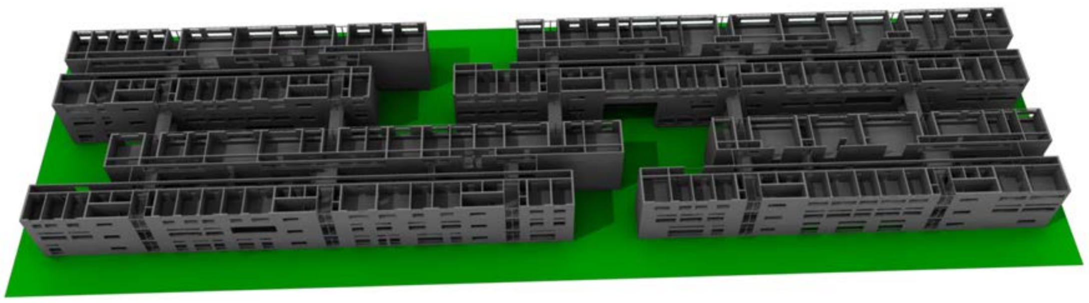


Fig. 2.7 Reconstruction of the office buildings in the Inffeldgasse 16 at Graz University of Technology using split grammars based on axis-aligned boxes. Terminal symbols are just filled or invisible boxes, no terminal geometry is loaded. (image source: Hohmann et al. [HHKF10])

Leblanc et al. [LHP11] were one of the first to try to realize interiors with split grammars. Interiors, especially, are a very hard problem to describe with split grammars. Room arrangements can become quite complex by featuring several non-convex rooms which are hard to describe by continuous splits. An especially ambiguous problem of realizing interiors through split grammars, which is the interpretation of the walls, is briefly discussed in Section 7.1. Leblanc et al. use split operations for façades and connections for decorations placements (like doors and windows). To realize the space partitioning of the interior Boolean operations are used on shapes. They, however, only touched the underlying problem by providing examples of low to medium complexity. Our research group also tackled this problem, which is – among other problems – discussed in Section 4.2.

Interactive Split Grammar Systems. The split grammars presented so far all heavily rely on scripting languages to produce their results. Artists with little computer science background may find it hard to use such tools. To simplify the process and make coding less mandatory for modeling with shape grammars, several interactive shape grammar systems have emerged to ease the modeling process.

The first to provide a real-time visual editing system was Lipp et al. [LWW08]. They provide visual representations for their operations, such as split operations, where the dividing planes are visualized. These elements can then be clicked and dragged for interactive editing of the grammar. Furthermore, to represent the hierarchical structure of the model, a tree-view is utilized in the graphical user interface. A similar system, with the main focus on façade modeling, has been presented by Krispel et al. [KHF10].

Recently, Krecklau et al. [KK12] introduced three-dimensional manipulators for any parameter in a procedural split grammar model. To overcome the massive amount of visual representations of parameters they use camera views to either concentrate on large scale parameters or detail parameters.

Patow [Pat12] introduced another paradigm for editing shape grammars. Instead of writing code, or editing directly at the three-dimensional model, he focuses on providing user friendly tools to edit the underlying graph, which represents the model generated through the grammar. He also provides methods for model verifications and full model editing through graph rewriting systems. Musialski et al. [MWW12] introduced a system in which modeling of façades is done based on photographs. They incorporate photograph analysis techniques to help and guide the user by providing an automatic initial placing of splitting lines. The GML Compositor, a tool that features a direct manipulation interface that has been created by our research group, is presented in Section 4.4.

2.3.5 Graph Grammars

Shape grammars usually operate in a context-free domain, but it is often necessary to take the immediate neighborhood into account. In shape grammars dependencies are usually expressed by a hierarchical tree data structure, but if more sophisticated dependencies are present, a tree is not sufficient anymore. This information flow problem is inherent, for example, in Gothic churches, where pillars, which are all at the same level in the hierarchy, need to be connected to ensure a structurally sound building. However, such constraints do not always need to involve structural design. Especially in churches, there are also constraints across the whole building in terms of design, or room layouts. Such context-sensitive dependencies can ideally expressed by graphs (see Section 3.2.1).

In comparison to shape grammars, graph grammars are a more general and powerful concept. Replacement rules replace a sub-graph, which feature labeled nodes and edges, and replaces it with another graph. These rules are mostly context-sensitive. However, finding a given sub-graph within another graph is a generalization of the maximum clique problem, which is NP-complete. Therefore appropriate data structures are necessary to improve the matching.

Graph rewriting systems are very common in software engineering to layout algorithms, but their applications in the context of visual computing have been sparse. Graph grammars have been used so far on meshes, which are graphs themselves, to realize basic modeling operations such as for example an extrude operation. Dormans et al. [Dor10] presented an application of a graph grammar to generate meshes. Based on an abstract graph definition of a (botanical) tree the corresponding mesh can be generated by applying the graph grammar to the acyclic graph describing the tree.

Christiansen et al. [CB12] use graph grammars in combination with shape grammars for level generation for action adventure games. A level of a game (or dungeon) is represented by two different structures: the *space* and the *mission*. The space is the geometrical layout of the level, which can be abstracted to a network of nodes (rooms) and their connections. The mission, on the other hand, is the series of tasks the player has to complete to progress to the end of the level. The mission can be represented by a directed graph in which a completion of a task makes other, new tasks available. There is a mapping between mission and space, but those two structure are in general independent because the same mission can be mapped to other spaces too. They analyzed famous action adventure games, such as games from *"The Legend of Zelda"* series, to define an alphabet of tasks (for example: "find key", "open lock", "defeat boss", etc.) that are essential to action adventure games. Based on these they designed a graph grammar that is able to generate missions and additionally provide a shape grammar that interprets the mission to generate an appropriate space.

Grasl and Economo [GE10] utilize a graph grammar to automate the Palladian grammar by Stiny et al. [SM78]. Later they improved their approach to create a system [GE11], which implements classical shape grammars (see Section 2.3.3) using graph grammars. Through the graph representation they can efficiently extract shapes that are generated by other overlapping shapes and can apply rules to them. Their system, however, handles all rules symbolically and no visual editing tools for rules are provided.

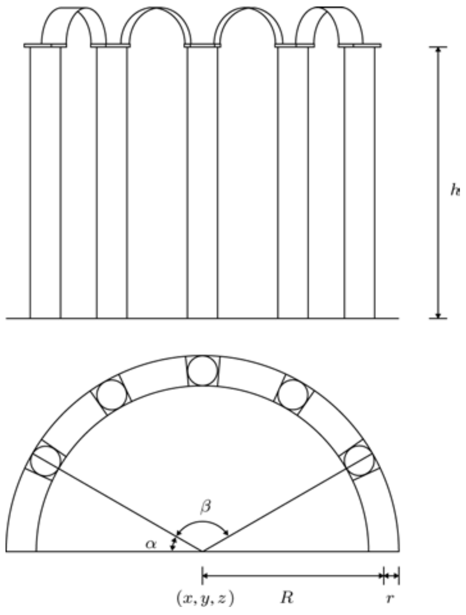
2.4 Inverse Procedural Modeling

Recently, inverse approaches in procedural modeling have become very popular. However, inverse problems are often ill-posed due to ambiguities and uncertainties and are therefore, in general, more difficult to solve. There are a lot of different approaches to inverse procedural modeling, such as fitting procedural models to three-dimensional point clouds, identifying shape grammars for rebuilding façades from photos, or finding repetitions in meshes for generating a procedural model. In general there is always the problem that some low-level shape is given and the procedural description that generates it is unknown. Therefore, a forward procedural modeling task is always a prerequisite for inverse procedural modeling. With a sufficient detailed procedural model given, the inverse task is responsible for finding the right parameters for fitting the procedural model to the available data. However, due to nature of inverse problems often being ill-posed, typically there is no unique best solution. It may be even so that there is no solution that generates an acceptable result.

2.4.1 Generative Surface Reconstruction

As before mentioned in Section 1.4.1, generative surface reconstruction focuses on understanding specific shape spaces through inductive reasoning. Once the shape is understood, inverse techniques can be applied for specific reconstruction tasks.

Already in 1999 Ramamoorthi and Avro [RA99] applied inverse techniques in the procedural modeling domain. In their work they generate short programs for describing railed, lofted, or swept surfaces from curves that are generated from scanned geometry. Ullrich and Fellner [UF07] use a genetic algorithm to find the best fit of a family of procedural models to a three-dimensional point cloud obtained from three-dimensional scans. Robustness against missing parts is important and is solved by a smooth localized distance metric for the one-sided distance between the procedural model and the point cloud. In 2008 their approach was further improved by the use of spatial data structures [USF08]. Ullrich et al. fitted a parametric model of an arcade (represented by a sequence of arches) to a three-dimensional scan of a cathedral in Pisa (see Figure 2.8). Further research by this group [UF11a] shows that using a probability distribution instead of a concrete shape for the formulation of the procedural model improved the early stage fitting performance. The later fitting steps for fine details are accelerated by using automatic differentiation (taking the derivative of a program) of the procedural model. Furthermore, the error to be minimized – the distance between the input data and the fitted model – can be estimated reliably by a constant number of samples [UF11b]. This number is independent of the complexity of the model as long as it is above a certain threshold. Ullrich et al. [USSF11], furthermore, provide an application for shape segmentation. The input mesh is assigned semantics during the fitting process. An offset texture is provided for the rough procedural model to approximately resemble the actual input mesh. The offsets are applied using displacements in a shader. Andrews et al. [AJS12] provide a system where they extract a pliable geometry description out from unstructured input data, such as meshes. They then apply forward modeling techniques to allow the extracted geometry to be further edited.



(a)

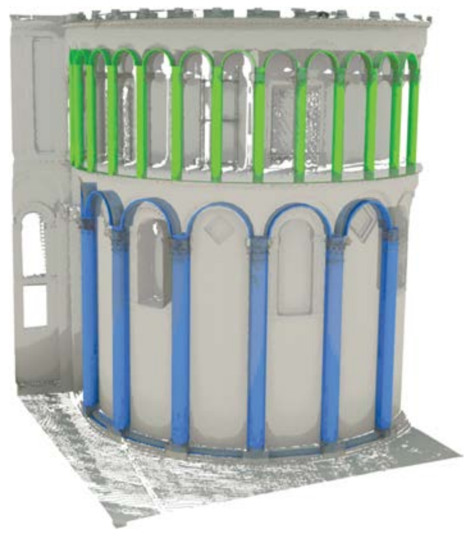

(b)

Fig. 2.8 The parametric description of circular arcades (a) used to fit generative models to a laser scan of the Duomo of Pisa (b). This circular arcade description features nine parameters: the origin (x,y,z), two radii R and r for the alignment and the columns, the number of columns n, an offset angle α, an opening angle β, and finally the height of the columns h. For this parametric description two instances can be found in the laser scan. (image source: Ullrich et al. [USF08]).

2.4.2 Grammar-Based Inverse Procedural Modeling

Aside from urban reconstruction, which will be discussed in Section 2.4.3, grammars are utilized in domains like detection of symmetric and repeated parts or contour fitting of procedural models.

Controlling grammar-based approaches (see Section 2.3), however, is often difficult. Slight changes of the grammar can result in unanticipated changes in the produced geometry. The recursive behavior of grammars is at fault for this unexpected behavior. The Metropolis approach from Talton et al. [TLL*11] solves this issue by turning it into an inverse problem. The desired outcome can be sketched by a contour which is used to generate a context-free grammar that fits objects, such as trees, buildings and cities in that contour.

Similarly, Bokeloh et al. [BWS10] take synthesizing context-free shape grammars as their goal. In a given input shape, they search for partial symmetries so that they can derive building blocks that can be recombined to generate new models out of parts of the input model (see Figure 2.9). Consequently, based on one model they span a design space of all models that can be generated by combining parts of this model. They further pursued their goal of parametrizing arbitrary input shapes by detecting partial symmetries and repetitions. In their paper in 2011 [BWKS11] they used free-form deformations to automatically adapt the model structure. Discrete (repetitions of elements) and continuous regularities (straight lines) are maintained while adapting the model through user-defined handles. However, sometimes undesirable distortions are introduced in the results. In 2012 they presented another approach in high-level shape editing [BWSK12] that does not suffer from the limitations of their publication in the year before. The input shape is analyzed for regular patterns and possible degrees of freedom are derived from that. Adjacent patterns are linked using linear constraints leading to a huge, but sparse linear system to be solved. Modifications of patterns always lead to a consistent shape in their representation.

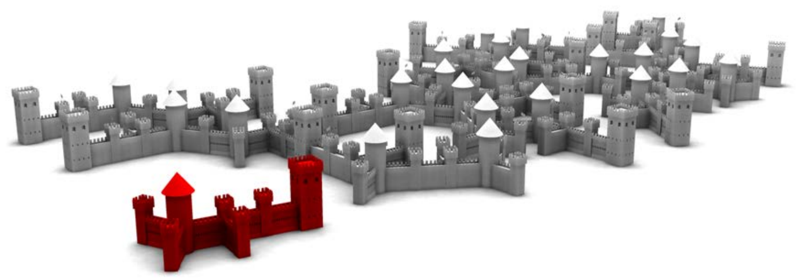


Fig. 2.9 Based on the input shape (red) Bokeloh et al. [BWS10] synthesize new variations (grey) with the help of detecting partial symmetries. Detected symmetric parts of the input shape are recombined to create variations.

A related approach by Lau et al. [LOMI11] analyzes furniture models available on the internet. Using shape grammars they partition these static models into parts and connectors necessary for fabrication. Application of the shape grammars also add inside parts, which are usually not present in static models.

2.4.3 Inverse Procedural Modeling for Urban Reconstruction

Urban reconstruction is a very prominent topic in inverse procedural modeling. A lot of approaches use shape grammars (see Section 2.3.3) because they are ideal for the analysis, description and construction of architectural elements, however, there are also approaches that do not use a grammar-related approach. Musialski et al. [MWA*12, MWA*13] provide an overview on urban reconstruction techniques in their recent survey and state of the art report. Low resolution building reconstructions in applications like Google Earth [Goo13a] or Bing Maps Platform [Mic13a] motivate the development of inverse procedural modeling of buildings and building parts from (aerial) image data.

Inverse Façade Reconstruction. One of the first attempts to replace low-resolution textures from aerial imagery to represent façades has been done by Müller et al. in 2007 [MZWVG07]. They use image analysis on a single view of a façade to meaningfully subdivide the corresponding photograph. Furthermore, they detect repetitions of façade elements (windows and doors) to extract shape grammar rules, which produce a three-dimensional model of the façade upon execution. This, however, is not done fully automatically. For an image with strong perspective effects of the target façade they can provide a first automatic approach [VGZVdBM07]. Formulating rule applications as statistical interference problems and using numerical approximation methods is a common approach to this problem. In the field of façade reconstruction such methods were first used by Alegre et al. [AD04] in 2004. *Reversible jump Markov Chain Monte Carlo (rjMCMC)* methods are another kind of such stochastic approaches and are used by Ripperda and his colleagues in several papers [RB07, Rip08a, Rip08b, RB09]. They aim to derive structural description of façades from range and image data automatically. Koutsourakis et al. [KST*09] use split grammars and a *Markov Random Field* to generate three-dimensional models out of rectified images from façades. Teboul et al. [TSKP10] use a context-free grammar to partition rectified building façades into semantic image segmentations. The shape grammar in use is, however, limited to a simple style of split combinations. They have applied their approach to façades that contain highly regular structures, namely Hausmannian and skyscraper-style façades. In contrast to their approach, Riemenschneider et al. [RKT*12] recently used *irregular lattices*, which preserve the logical structure of a façade. Their approach is better suited to represent less regular, but still highly structured façade layouts. Their matching algorithm supports detecting of repetitions and symmetries in façades (see Figure 2.10).

Inverse Building Reconstruction. Inverse procedural methods are also used to re-model or detect entire building models from different available data, such as photographs or three-dimensional point clouds. Cornelis et al. [CLCG06] combine algorithms for city reconstruction using a video recorded by a car and a car detection algorithm using a cognitive loop to overcome the individual limitations. Cars could be matched reliably using the three-dimensional information obtained by the reconstruction algorithm. This way, cars could be removed from the reconstruction and be replaced by actual car models. Mathias et al. [MMWVG11] apply inverse procedural modeling to buildings and landmarks. Through *structure from motion* techniques they acquire a three-dimensional point cloud from a series of images. They use a machine learning-based asset detector that aids the shape grammar interpreter, which drives the reconstruction process. Their use cases are crumbled Greek Doric temples. Toshev et al. [TMT10] extract buildings from raw three-dimensional point clouds and parse them in geometric and semantically meaningful parts via a grammar. They view a building as a tree which

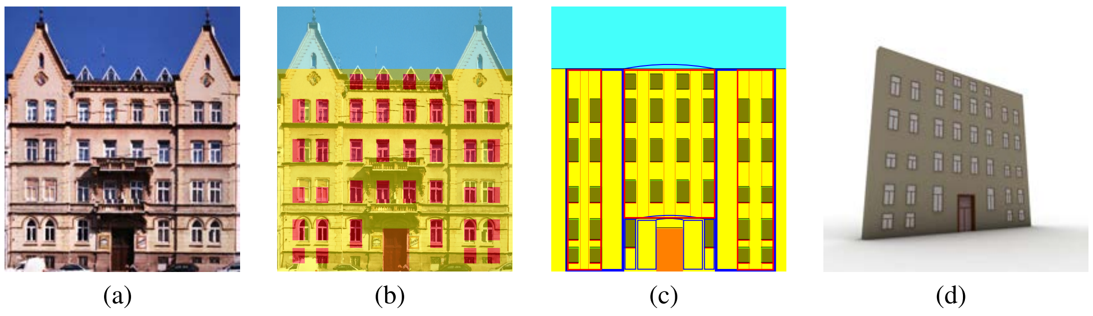


Fig. 2.10 From an orthophotograph of a façade (a) pixels are segmented to belong to different classes such as windows, doors, walls and sky. Based on this segmentation a grammar refines the result (b). Symmetry and repetitions are detected in the parsed structure (c) to ensure coherence throughout the related parts. The parse tree can then be automatically transformed into a three-dimensional model (d). (image source: Riemenschneider et al. [RKT*12])

consists of neighboring volumetric parts that are next or above to each other and are covered by planar patches representing the roofs. To extract the planes that make up these parts they use a *RANSAC*-based plane detection algorithm. Based on the *Manhattan-world assumption*, namely that in each scene there is a predominance of three mutually orthogonal directions, Vanegas et al. [VAB10] produce a single, coherent, and complete model of a building using calibrated aerial textures. They use a grammar to sequentially refine a coarse initial model (e.g., a box). Nan et al., on the other hand, have developed their own method called *smartBoxes* for reconstruction of building based on sampled point clouds. In their system the user loosely defines and manipulates building blocks, which then automatically snap to their proper location adjust their size and orientation. Li et al. [LZS*11] have combined photographs and three dimensional LiDAR scans to drive this idea further. Through combination of two-dimensional and three-dimensional data they can reduce they noise and remove outliers from the point clouds. Their information transfer is bidirectional: depth information is transferred to the photos and afterwards information of the augmented photograph is transferred back. Through several assumptions on a planar façade, free form buildings are ill-suited for their approach.

2.5 Specific Procedural Modeling Approaches

Several techniques for procedural modeling have been explored by now, however, there are still some methods worth mentioning. This section focuses on methods which did not fit into the sections before.

2.5.1 Procedural Modeling of Cities and Buildings

Procedural Modeling of East Asian Architecture. A work worth mentioning in this regard is the unifying mathematical model that is able to describe the structures of East Asian buildings by Tenou et al. [Teo09]. They describe different building types, such as palace halls, commoner houses, temples, pagodas and pavilions of Chinese, Japanese and Korean culture. They introduce a graphics modeling library supporting this large range of traditional East Asian styles and structures. Similarly, Huang and Tai [HT13] provide an interactive procedural modeling tool for the generation of Chinese ting.

Procedural Modeling through Floor Plan Extrusions. A common way to model entire buildings in computer-aided design software is to generate a floor plan and extrude it for each separate floor. In the architectural point of view, a three-dimensional representation of a house complete with interior is often used for simulations like light and heat propagation before construction. For virtual applications like computer games, on the other hand, just the visual appearance and soundness of the produced building matters.

Yin et al. [YWR09] provides a survey of different floor plan extrusion techniques up to 2009. They look into computer-aided design-based methods as well as methods that take rasterized images of floor plans as input, where algorithms for denoising and text removal are used to extract the actual floor plan. Horna et al. [HMDB09] propose a semi-automatic method for generating three-dimensional buildings from two dimensional vector plans. They utilize a formal description of constraints to provide a generic representation of geometry, semantics and topology of architectural indoor environments. *Generalized maps (G-maps)* are used as a topological basis to represent their buildings through volumetric cells.

Merrell et al. [MSK10], on the other hand, do not rely on predefined building plans. Through a set of high-level specifications, like the number of bedrooms, or specific room adjacencies, they generate an architectural program. These programs specify each room in the building together with its adjacencies and approximate dimensions through a Bayesian network trained on real world data. A detailed set of floor plan is generated using these architectural programs through statistically optimizing over the space of possible floor plans. These set of floor plans can then be used to generate a full three-dimensional building using extrusions. Their method, however, has problems with roof parts overlapping windows in the upper floors.

In contrast, Kelly and Wonka [KW11] focus just on the exterior shell of the building. A profile describing the course of the wall and the roof can be defined for each side of the ground plan of the building. During the extrusion each side of the building follows the path defined in the associated profile. A *straight skeleton* [AAAG95, EE98] algorithm is utilized to correctly merge the different profiles. Elements like meshes of windows and chimneys can be placed afterwards by specifying their locations in the ground plan and the wall profiles (see Figure 2.11).

Procedural Modeling for Street Networks and City Planning. For generating realistic cities the underlying street network needs to be designed too. Chen et al. [CEW*08] propose a method based on tensor fields. For a given water map, the user generates a tensor field that guides the generation of the street graph. The tensor field marks the directions in which the main roads are placed. The tensor field can still be edited before placing the minor road pattern. Parks and building geometry is added in the last step.

Vanegas et al. [VKW*12] offer a method to create parcels within city blocks procedurally. They propose a combination of two different subdivision techniques that generate spatial parcel configurations with a high similarity to those observable in real cities. They also provide an editing system where roads as well as block parameters can be tuned interactively. Their algorithms have been implemented in Esri's CityEngine [Esr13].

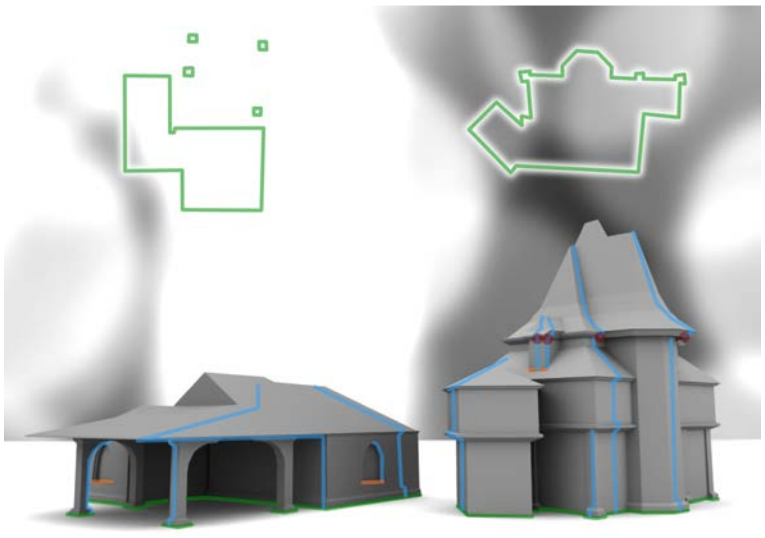


Fig. 2.11 In the work of Kelly and Wonka [KW11] buildings are realized by profile extrusions using a straight skeleton algorithm. Profiles can be defined for each edge of the ground layout polygon. Architectural elements like windows or dormers can be placed by specifying their location in the ground plan as well as the profile.

2.5.2 Procedural Modeling of Trees and Plants.

L-systems (see Section 2.3.2) are not the only technique used in procedural modeling of trees and plants. This section intends to give a rough overview on different approaches to procedural plant modeling.

Computer modeling of trees and plants can be traced back to the origins of two different approaches. On the one hand there was Ulam [Ula62], who considered trees as self-organizing structures, where branching patters emerge dependent on competition for light and space (*space-oriented*). On the other hand, Honda [Hon71] considered trees as explicit recursive structures (*structure-oriented)*. These structures are characterized by parameters such as branching angles or internode lengths. These two different views led to a multitude of papers on both sides.

Structure-Oriented Methods for Procedural Plant Modeling. Honda's view is ideally suited to be realized using recursive generative algorithms which have been used by Aono and Kunii [AK84], Bloomenthal [Blo85], and Oppenheimer [Opp86]. These early recursive models only allow control over local features such as branching angles. More control is supported by the approach of Reeves and Blau [RB85] and has been further developed by Weber and Penn [WP95], Lintermann and Deussen [LD99], and Prusinkiewicz [PMKL01]. In these approaches features like tree silhouettes and density of branches can be specified which leads to highly realistic models. Especially the approaches from Lintermann and Deussen [DL97, LD99] received much attention for developing the *Xfrog* [Xfr13] modeling technique. For all these representations, however, it is not possible to dynamically react to changes in their environment.

Space-Oriented Methods for Procedural Plant Modeling. In contrast, Ulam's idea has been first expressed using two dimensional cellular automata and has been extended to three dimensional voxel spaces by Arvo and Kirk [AKC88], Greene [Gre89], Beneš and Millan [BM02], and Palubicki et al. [Pal07, PHL*09]. The aspect of competition for resources is the guiding topic of these papers. De Reffye et al. [dREF*88] present a description of plant modeling adhering to botanic laws. Their model is based on the different fates of buds: some buds may begin to bloom and become fruits, where others produce new branches or just abort. Afterwards, others also incorporated the different fates of buds into their representations [PJM94, MP96, BM02, CSR*08, PHL*09]. These are some the most physically-faithful growth models for plants. They incorporate influences of the surrounding environment as well as biological mechanisms such as competition for resources, such as light, space and water. Pirk et al. [PSK*12] present a system where trees interactively adapt to their surrounding environment. Their system allows import of any tree model independent of the system it has been created with. They analyze the tree structure to extract the tree skeleton and leave clusters on which calculations are made (see Figure 2.12).

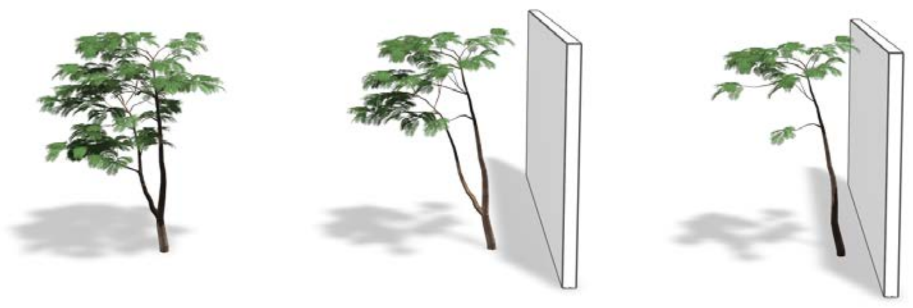


Fig. 2.12 Trees imported into the system of Pirk et al. [PSK*12] are analyzed and then appropriately pruned and bent based on obstacles that are introduced to the surrounding environment in which that tree grows.

2.5.3 Procedural Shading and Textures

Procedural techniques are not only used to generate three-dimensional content. Procedural shading techniques are used to decorate shapes in an appropriate realistic way without repetition and resolution artifacts from texturing. This sections indents to give a rough overview of this domain without going into too much detail.

Procedural Textures Learned from Images. Procedural textures are used to ensure a constant transition within the texture without visible borders. This is realized through stitching together small patches of existing images in the work of Efros and Freeman [EF01]. Hertzman et al. [HJO*01] process pairs of images, where one image is purported to be a filtered version of the other. The filtered image is used as training data to learn the applied filter. This learned filter kernel is then applied to the second image. This process allows many applications among which are:

- *Traditional image filters*, like blurring or embossing.
- *Super-resolution filters*, where a high-resolution image is inferred from a low-resolution source.
- *Texture transfer*, where a texture from one image (an oil painting) is transferred to another image (a photograph).

Three-Dimensional Procedural Textures. To realize highly realistic models made of wood or marble, three-dimensional textures are used to simulate the process of production. These textures are described by mathematical means. To provide more realism, a three-dimensional noise texture is applied to the texture.

Location-Dependent Procedural Textures. Procedural textures are, furthermore, used to decorate three-dimensional content corresponding to their shape and environment. Procedural textures for mountains, for example, have several parameters that may change based on the elevation of the surface; on higher parts there will be less grass and a rockier surface, and even higher up there will be snow and ice. The technique of Lasram et al. [LLD12] visualizes all such possible appearances in procedural textures in a single image with limited resolution. Legarkis et al. [LDG01] concentrate on cellular texturing of architectural models. Textures of cellular patterns, like in brick walls, often do not match at corners. They present a technique based on three-dimensional cell entities, which are placed at the surface of the model to texture.

2.6 Structured Shape Editing

The domain of structured shape editing is closely related to procedural modeling. The goal of structured shape editing is to edit existing shapes and reassemble parts obtained from a collection of objects into new shapes. This section will provide a rough overview over the approaches in the domain of structured shape editing of man-made shapes. A more detailed survey where key concepts and their methodological approaches are summarized and organized is available in the state of the art report by Mitra et al. [MWZ*13] titled *"Structure-Aware Shape Processing"*.

Structured shape editing algorithms mainly consist of two phases: an *analysis phase*, where structural information is extracted from input data, and a *processing phase*, where the extracted information is used to create new shapes.

Analysis Phase. The analysis phase is characterized through the task of understanding shapes and structures. At the core of this phase lies the part analysis, which is a classical segmentation problem. Detecting symmetry plays a very important part when processing man-made shapes. Inverse procedural modeling techniques as well as shape grammars help in this task.

In general, a shape is a collection of *parts*, their *parameters* and the *relations* between them. A part is a logical entity of a shape with semantic importance. Parts are not necessarily disjoint and can therefore overlap. Each part has a finite set of parameters that define the appearance of that part. A part in this sense does not need to be a surface patch resulted from the segmentation; it is more an abstraction of a region of a shape and acts as a proxy for that semantic part. Relations between objects define how parts, together with their parameters, correlate. The underlying structure of the shapes is described by these relations, which are realized by constraint function that evaluate to zero for valid shapes.

There are mainly three different ways for identifying parts, parameters and their relations:

User Specified Segmentation.
In the simplest case the user specifies parts, parameters and constraints. Through procedural modeling custom parameters can be defined by scripts or shape grammars. Constraints can also be specified and be checked automatically to validate the constructed shapes. This is especially common within computer-aided design and sketch interfaces. Shapes described this way can adapt to various circumstances that cannot be foreseen by automatic methods.

Fixed Model Related Segmentation.
For models with properties known a priori, a fixed segmentation model can be constructed and applied to obtain the necessary parts. An example for this is the work of Bokeloh et al. [BWS10], where just static parts are extracted from a fixed model (see Figure 2.9). There are many methods where the set of parameters are fixed beforehand. The arrangement of parts is done, for example, using rigid transformations or the usage of fixed geometric primitives, which feature a well-known set of parameters. The same can be said about relations. Physical constraints such as balance (position of the center of mass), or connectivity of parts are known beforehand and can be maintained while editing. Umetani et al. [UIM12] use such constraints for furniture design.

Segmentation Learned From Data.
Lastly, instead of a model based on a priori assumptions, a machine learning approach is used to extract all information. This method is certainly the most general, but also has prerequisites such as the requirement of a set of training data. These methods can either be supervised (with user annotations) or unsupervised (completely autonomous). Parts are usually extracted using segmentation methods. For parameters and relations the degrees of freedom need to be extracted from the training data. Common models for learning parameters are dimensionality reduction techniques, like the *principal component analysis (PCA)*. Constraints can be for example learned with the help of shape grammars, which are based on local similarity (like in the work of Bokeloh et al. [BWS10]).

It is not necessary to use the same method for for each identifying task. Different methods can be combined for the determination of parts, parameters and their relations. Kalogerakis et al. [KCKK12], for example, use manually labeled input shapes with fixed parameters, but extracts the relations from these input shapes for the recombination of the labeled parts.

Processing Phase. The processing phase is for editing existing shapes and the synthesis of new shapes with the data acquired from the analysis phase. For editing existing structures free-form deformations [SP86, Coq90] are a commonly used tool because they preserve structure. The iWires system by Gal et al. [GSMCO09] detects crease lines (called wires) in triangle meshes, which define the characteristics of the shape. The user can generate handles to deform the wires, which also deforms the shape. Afterwards the system tries to restore the mutual relationships (parallelism, orthogonality, symmetry) between the wires to generate a shape that meets the user constraints and better preserves the original structure.

New shapes can be generated by assembling different parts from input shapes. These approaches are also referred to as *data-driven methods*. Single input shapes as in the work of

Bokeloh et al. [BWS10] (see Figure 2.9), as well as shape collections can be used. Xu et al. [XZCOC12] use concepts from evolution theory to generate new models starting from a set of structured models belonging to the same class (e.g. chairs in Figure 2.13(a)). As mentioned before, Kalogerakis et al. [KCKK12] recombine parts of pre-segmented shapes with the help of probabilistic models to generate new ones (see Figure 2.13(b)).

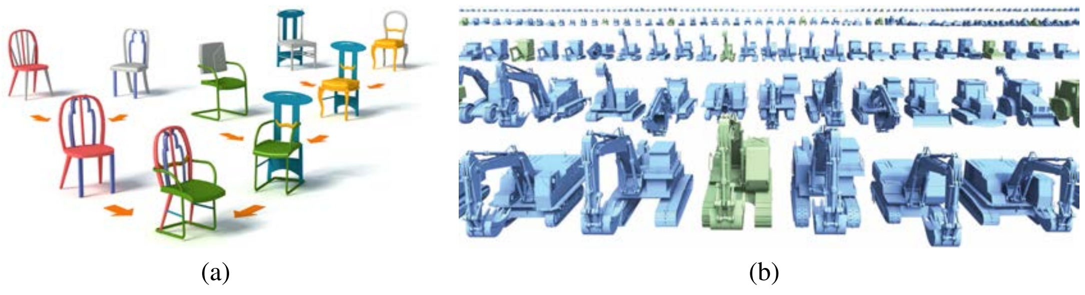


Fig. 2.13 Examples for data-driven structured shape editing methods. In the work of Xu et al. [XZCOC12], evolutionary processes lead to the generation of a new chair (a), that shares parts with chairs from an initial generation. Kalogerakis et al. [KCKK12] use probabilistic methods to recombine parts from pre-labeled existing shapes (green) to generate new shapes (blue) (b).

Finally, reconstruction tasks can also be performed from structured shape editing techniques. Shen et al. [SFCH12] retrieve and assemble parts taken from a shape collection to reconstruct shapes from three-dimensional scans taken with the Microsoft Kinect Sensor [Mic13b].

Future Challenges for Structured Shape Editing. As mentioned before, so far work in this field has only be demonstrated on small to medium sized data sets of the same object family. The scalability of these approaches has not been sufficiently investigated so far, as has been stated in the report of Mitra et al. [MWZ*13]. Furthermore, for all the data-driven approaches there is also the problem of the sparse availability of high quality shapes and shape collections. To provide high quality results, these algorithms need high quality input.

Training sets that are used for the presented methods are in the most cases manually labeled. Results are influenced positively by a qualitative manual labeling, however, this approach does not scale when larger data sets are used.

Structured shape editing is not considered to be a variant of procedural modeling, but the combination of both methods is promising. With parts that are procedural models itself or are extracted from such, many issues from current structured shape editing can be solved.

Synopsis

In this chapter I presented fields and ways of application of procedural techniques with a focus on procedural modeling. The grammar-based approaches are of particular interest for this thesis. Split grammars are the established state of the art in modeling and generation of digital cities and architecture, which is one of the representative areas this thesis is focused on. Split grammars follow a coarse-to-fine modeling approach build upon rule-based replacements. Labeled shapes are partitioned into smaller parts by rules that match the assigned label to increase the detail of the model. This approach makes it ideal to describe hierarchical structures, such as buildings or façades. One major drawback of split grammars is their context-free nature, which makes it hard to connect shapes from within the grammar. Split grammars have been subject to many extensions that address limitations or enhance their expressiveness, among which are the extensions featured in this thesis.

Another important message of this chapter is the necessity of procedural forward modeling techniques. Inverse procedural modeling is very popular technique for the reconstruction of shapes based on point clouds or photographs. However, these techniques require procedural models, as well as modeling techniques, for fitting shapes to the available data. It is often neglected that a procedural analysis of the shape is necessary to make these inverse techniques work.

Chapter 3
Digital Shape Representations for Procedural Modeling

Contents

Abstract. Procedural modeling techniques create shapes in various ways. But before one can utilize these techniques it is necessary to know, understand, and distinguish different low-level shape representations and structuring mechanisms, which are then used to produce shapes procedurally. These low-level shape representations include first and foremost different representation of geometry, like meshes, subdivision surfaces, or implicit surfaces, which all can be created procedurally through sets of well-defined operators. Shapes for industrial parts are often described by combining several low-level primitives together. These primitives can easily be parametrized and thus the whole combined shape is parametrized too. These structuring mechanisms combine low-level representations to describe hierarchies, dependencies and relations within procedural models. This chapter focuses on these low-level shape representations and structuring mechanisms to provide the theoretical foundation for the chapters to come.

3.1 A Hierarchy of Shape Representations

To differentiate between established shape representations, this section will assign them into three levels based on their purpose and usage. These levels are: *low-level shape representations*, which are used to describe the surface of shapes, *structuring methods*, which take shapes expressed by the methods before and set them into a context, and, finally, *procedural methods* that are used to describe and create the above mentioned structures. This hierarchy provides an overview on how established methods can be combined to create different shapes. In the following, each of the three levels of this hierarchy will be explained.

Low-Level Shape Representations. Many different shape representations have been invented during the years of research in the field of computer graphics. These representations are just different ways to describe shapes and are based on the available data as well as a specific application area. It is important that none of these representations have been invented with procedural modeling in mind, but all of them can be used to create shapes procedurally. There exist a wide range of *low-level* shape representations for different domains, among which are

Surface Sampling.
The sparsest representation of three-dimensional data is a simple point cloud, which samples of the outer surface of an object. In point clouds no information on connectivity of the single vertices is stored. Point clouds are often used in digital documentation processes and are commonly obtained by a laser scanner in a very high density.
Surfaces can also be sampled through a connected set of points, which span planar regions on the shape's surface. These so-called polygonal meshes are the most commonly used representations. These representations include many different types, among which are for example triangle meshes and quad meshes, but also general boundary representations, which include faces of arbitrary degree. Polygonal meshes are the topic of Section 3.3.

Free-Form Surfaces.
Curved surfaces need a great number of vertices and faces to be represented exactly by polygonal meshes. Free-form surfaces, on the other hand, are able to describe curved surfaces in a mathematical way and are therefore more exact than mere polygonal meshes. Non-Uniform Rational Basis Splines (NURBS) are one of the most commonly used free-form surface representation. NURBS curves and surfaces are both generalizations of *Bézier splines* and *B-splines*. Subdivision surfaces, on the other hand, refine a coarse polygonal mesh over many iterations to reach a smooth surface. The Catmull-Clark subdivision procedure for subdivision surfaces is examined in Section 3.4.

Volumetric Shapes.
Instead of sampling the surface of a shape, volumetric shape representations use volumetric entities to approximate the volume of a shape. Convex polyhedra, for instance, describe shapes by the intersection of half spaces. Naturally, only convex shapes can be described this way. To express non-convex shapes, sets of convex polyhedra are necessary. This shape representation is explained in Section 3.5. Other representations approximate a volume through the union of a set of different sized spheres. This method is called "union of spheres". In the metaballs approach, ball-like shapes interact with each other by merging together over time. This way, organic surfaces can be described easily.

This list of domains and corresponding shape representations is not intended to be complete and it will also grow with the years of research to come. The important task is to make these shape representations accessible to a procedural modeling approach. This is done by a set of operators to describe the generation of these shapes. In his Ph.D. thesis, Havemann [Hav05] demonstrated this in case of meshes, which will also be reviewed in Section 3.3.2. If an operator interface can be provided in the case of meshes, other low-level shape representations can also be analyzed to see whether an operator interface can be provided for them or not. In this chapter additional to polygonal meshes (see Section 3.3), subdivision surfaces (see Section 3.4) and convex polyhedra (see Section 3.5) are explained to provide the theoretical background for the upcoming chapters. These explanations eventually lead to operator interfaces for all these low-level shape representations.

Structuring Methods. There are also methods that do not describe a shape itself, but can take shapes independent of their representation and process them further. This can either be by bringing them into relation with other shapes or by changing the overall appearance. The most prominent of these methods are scene graphs (see Section 3.6) as well as constructive solid geometry methods (see Section 3.7). In addition to these, free-form deformations (see Section 3.8), which alter the appearance of a shape, are discussed too. In this chapter an operator interface is provided for these structuring methods to make them available for the procedural approach.

Procedural Methods. At the highest level are the *procedural methods*. These are the most general methods because they characterize three-dimensional shapes semantically. A procedural representation cannot be visualized directly, but through execution it generates shapes, which can be described by low-level shape representations and may be set into context by structuring mechanisms. To create a procedural representation for the above mentioned structures and representations, the design of an operator interface is necessary. This set of operators need to encapsulate the generation and modification of these shapes.

All the methods presented in the preceding Chapter 2 belong to these procedural methods. The focus of the upcoming Chapter 4 will be entirely on the specific procedural methods that have been used and realized in context of this thesis. These methods include scripting languages, split grammars, and data flow graphs.

3.2 Foundations of Basic Data Structures

Before the chosen low-level shape representations can be presented, important theoretical background is reviewed to provide the information necessary to understand all concepts that are presented from this point onward.

3.2.1 An Introduction to Graphs

It is important to discuss the abstract mathematical structures named graphs because they are the foundation of for several of the low-level shape representations and structuring methods that have been mentioned above, in particular polygonal meshes (see Section 3.3). Consequently, procedural techniques that describe these representations need to offer operators, which (indirectly) build graph structures. A graph can also be used to represent interconnections within a set of objects. Graphs are, therefore, very important in the domain of procedural modeling because hierarchies, procedures, dependencies and thus structuring techniques, such as scene graphs (see Section 3.6) and constructive solid geometry (see Section 3.7), can be described as graphs. This section will cover the most important facts about graphs; further more detailed information on graphs in general is available in the book by Diestel [Die05].

Definition 3.1 *A **graph** G is a tuple (V,E), with V as a set of nodes (or vertices) and E as a set of edges. An edge $e = \{u,v\}$, $u,v \in V, e \in E$ describes a connection between two nodes. Nodes, as well as edges, may carry any label.*

For a graph $G = (V,E)$, two nodes u and v, $u,v \in V$ are called *adjacent* if there is an edge $e = \{u,v\}, e \in E$ that connects u and v. A *path* of length n is a sequence of n edges $e_0 = \{v_0,v_1\}, e_1 = \{v_1,v_2\}, \ldots, e_{n-1} = \{v_{n-1},v_n\}, v_0, \ldots v_n \in V, e_0, \ldots e_{n-1} \in E$ that allows a transition from a node v_0 to a node v_n. The *degree* of a node is the number of edges that involve that node.

Edges are used to traverse between nodes and can be interpreted differently based on the properties of the graph:

Connected Graphs.
If there are no disjoint graph components a graph is called *connected*, otherwise it is called *disconnected*. A disconnected graph can always be split into connected components.

Directed Graphs.
A graph $G = (V,E)$ can either be *directed* or *undirected*. If a graph is not marked directed, it is undirected by default. Undirected and directed edges can be distinguished by the notation. For undirected edges the set notation with curly brackets is used, but for directed edges parenthesis are used. In undirected graphs the edges $e = \{u,v\}$ and $e' = \{v,u\}$ are redundant because one single edge allows the traversal between the two corresponding nodes in any direction. For directed graphs, however, $e = (u,v)$ and $e' = (v,u)$ are different; e only allows to move from the node u to the node v, but not in the other direction.

Acyclic Graphs.
When for all nodes $v_i \in V, 0 \leq i < |V|$ of a graph $G = (V,E)$, no path that starts and ends in v_i exists, the graph is called *acyclic*. A *cycle* is a path starting and ending in the same vertex, where each two consecutive edges $e_i = \{u,v\}$ and $e_j = \{v,w\}, u,v,w \in V, e_i,e_j \in E$ in the path end, respectively start, in the same node v. In an undirected graph there is per default always a cycle of length two because edges are bidirectional.

Weighted Graphs.
Edges can additionally carry weights as attributes that might represent costs, lengths, or capacities. The values of these weights can, for example, be probabilities or just the Euclidean distance between the corresponding nodes. They can also be completely unrelated to the position of the nodes, which means that the *triangle inequality*, which states that for any arbitrary triangle the sum of the lengths of any two sides must be greater than the length of the remaining side, does not need to be fulfilled by the weights.

Multigraphs.
Multigraphs feature multiple edges between two nodes. Even *loops*, edges $e = \{u,u\}$ that connect a node with itself, may be permitted based on the application.

Hypergraphs.
In hypergraphs an edge connects instead of two vertices two different sets of vertices with each other.

Infinite Graphs.
In general, for a graph $G = (V,E)$, V and E are finite sets. If one or both sets are infinite in size, the graph is called *infinite*. If not mentioned otherwise, graphs are always *finite* by default.

Graphs have many applications in mathematics and computer science. Based on the application, the used graphs have different properties. *Markov chains*, for example, are weighted directed graphs with probabilities as edge weights, and *finite automata* are examples for directed multigraphs with edge labels.

An important graph type is the so-called *tree*, which is ideal to describe hierarchical structures.

Definition 3.2 *A **tree** $T = (V,E)$ is a connected acyclic graph where any two nodes are connected by exactly one single path. Nodes of degree one are called **leaves**. A unconnected union of trees is called a **forest**.*

Definition 3.3 *An **oriented tree** $T = (V,E)$ is a variant of a **directed acyclic graph (DAG)**. The underlying undirected graph is a tree. In such an oriented tree there is one designated node that has only outgoing edges, called **root node**. Nodes with only incoming edges are **leaf nodes**. A node is called a **parent node** for all nodes that are adjacent to it. These nodes are the **child nodes** of their parent.*

There are special variants of trees, for example, *binary trees* where each node has at most two children. Directed acyclic graphs are utilized in many ways within procedural modeling. Prime examples here are scene graphs (see Section 3.6) and data flow graphs (see Section 4.4.1).

3.2.2 Topological Foundations of Meshes

Graphs can be used to approximate a three-dimensional surface. These so-called *meshes* sample the surface in question and contain – in addition to vertices and edges – faces, which are surrounded by a closed sequence of vertices and edges. This section will introduce topological foundations and features of meshes leading up to Definition 3.10 of a *valid* mesh. It is important to adhere to these features while designing an operator set to procedurally generate valid meshes (see Section 3.3) because otherwise these meshes may cause problems if further processed, for example for three-dimensional printing. A more in-depth discussion of this topic is available in the book by Botsch et al. [BKP*10] titled *"Polygon Mesh Processing"* as well as in the Ph.D. thesis of Havemann [Hav05].

Definition 3.4 *A **polygonal mesh** M is a tuple (V,E,F), where*

- *V is a finite set of three-dimensional points, called **vertices**,*
- *E is a finite set of **edges**, which connect two vertices, and*
- *F is a finite set of polygons called **faces**, which consist of vertices and the respective edges that describe the corresponding polygon. All vertices of a face lie within one plane.*

*The **valence** of a vertex is the number of edges incident to that vertex. An edge $e \in E$ is described by a tuple of two vertices: $(v,v'), v,v' \in V$. The **boundary** of a face $f \in F$ is described by a sequence of vertices and edges: $(v_0,e_0,v_1,e_1,v_2,\ldots,v_n,e_n)$ where $e_i = (v_i,v_{i+1})$, for $v_i \in V, e_i \in E, 0 \leq i \leq n$. Since this sequence is cyclic, it means that the index i is always taken modulo n, and the edge e_n connects vertex v_n to vertex v_0.*

Meshes feature different topological properties. This section will examine important topological mesh properties that will lead to the definition of a valid mesh. I will start with the property that defines whether a mesh is *closed* or not.

Definition 3.5 *A mesh is called* ***closed*** *if all edges corresponding to this mesh are incident to at least two faces.*

If an edge in a mesh has only one incident face it is called a *boundary edge*; this edge is consequently incident to a *hole* in the mesh, which is therefore not closed. A sequence of boundary edges is called a *border*. From here on only closed meshes are considered because each hole in a mesh can be filled by inserting additional faces. Consequently, each mesh that is not closed can be transformed into a closed mesh.

An important topological measure of a mesh is its *Euler characteristic*, or *genus*. The genus of a mesh is an integer value counting the number of topological holes in the mesh. So a sphere has genus zero, while a torus has genus one. Holes formed by boundary edges are no topological holes, so a cylinder with missing top and basement faces has still genus zero.

Another important topological property of a mesh is its *orientability*.

Definition 3.6 *A mesh is called* ***orientable*** *if and only if that mesh does not contain a Möbius band. Otherwise it is called a non-orientable surface.*

In an orientable mesh all edges incident to a face can be oriented in a consistent way. By following the orientation of the edges, a cycle around the boundary of the face is formed. Two adjacent faces incident to an edge traverse this common edge in opposite directions.

A Möbius band (see Figure 3.1(a)) is a surface with only one side and one border, leading to the fact that the interior and exterior cannot be distinguished. A solid object, however, has a clearly defined finite interior, infinite exterior, and the surface separating those. Other non-orientable surfaces are *Klein bottles* (see Figure 3.1(b)) and *real projective planes*. From this point onward only orientable surfaces are considered.

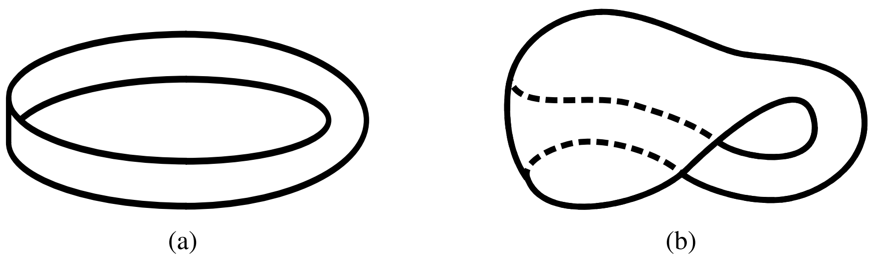

(a) (b)

Fig. 3.1 Non-orientable surfaces: the Möbius band (a) and a Klein bottle (b). These surfaces have only one side and one border, meaning that the interior and the exterior cannot be distinguished from another.

The following property of a mesh defines whether it is called *manifold* or not. This property is important to guarantee consistent mesh traversal without ambiguities.

Definition 3.7 *The* ***relative neighborhood*** *of a point on a three-dimensional surface is the intersection of the surface with an infinitesimal small three-dimensional sphere centered at that point.*

Definition 3.8 *A three-dimensional mesh is called* ***manifold*** *if and only if each point on the surface fulfills the following two constraints:*

- *First, the relative neighborhood of each point needs to be topologically equivalent to a two-dimensional open disc. This mapping between the relative neighborhood and the open disc has to be bijective and continuous.*
- *Second, the relative neighborhoods of two distinct points need to be disjoint.*

This definition can be expanded to arbitrary dimensions. An n-dimensional manifold is a topological space in which every point has a relative neighborhood that is topologically equivalent to an n-dimensional open disc. A mesh with that properties is therefore a *two-manifold* (see Figure 3.2).

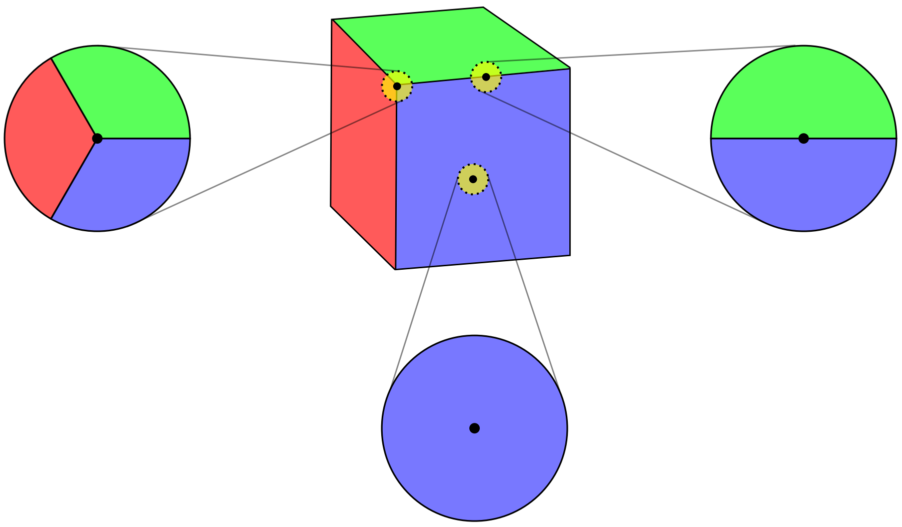


Fig. 3.2 For a two-manifold mesh the relative neighborhood of each point on the surface of the mesh needs to be equivalent to a two-dimensional disc. This is illustrated with points on the surface of a cube. The colored segments in the discs correspond to the equally colored faces of the cube.

The following properties, which are important for navigation on meshes, can be derived of the definition of a two-manifold mesh:

Manifold Edge Property:
Each edge is incident to exactly two faces.
Manifold Vertex Property:
For each pair of two faces incident to a vertex, a transition between those faces is possible by traversing to neighboring faces over edges incident to that vertex.

If those criteria are not met a mesh is called *non-manifold*. Examples for meshes violating those criteria are shown in Figure 3.3.

For a manifold mesh the *Euler-Poincaré formula* provides a topological invariant.

Definition 3.9 *For a manifold mesh $M = (V, E, F)$, let $v = |V|$, $e = |E|$, $f = |F|$ be the number of vertices, edges, and faces respectively. Furthermore, let s be the number of connected components (shells), h the number of topological holes (which is the genus), and r the number of edge rings (loops) in faces.*

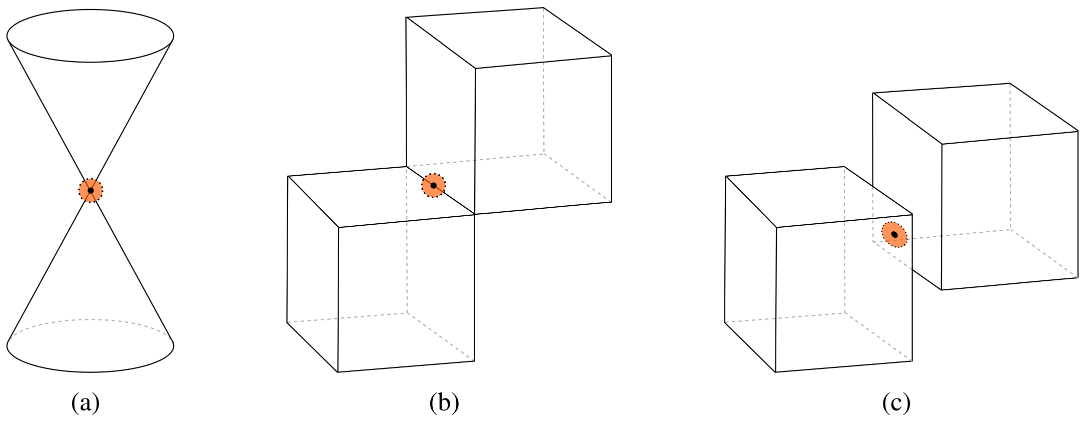


Fig. 3.3 Meshes that violate the properties of a two-manifold mesh. In a double cone (a) one vertex is part of both cones. The neighborhood of this non-manifold vertex cannot be embedded into a two-dimensional disc. When two blocks share the same or part of an edge (b) all points along the shared part are non-manifold. Sharing parts of faces (c) leads to the same problem. All points within the shared part of the faces are non-manifold.

*Then the **Euler-Poincaré formula** states that*

$$v - e + f = 2 \cdot (s - h) + r$$

This formula can be used to determine if the shape is topological invalid. If this equality does not hold, then something is wrong in the representation. However, for a topological valid shape it is necessary, but not sufficient that this equality holds.

Finally, this leads us to the definition of a valid mesh:

Definition 3.10 *A mesh is called **valid** if and only if it is two-manifold, closed, and orientable.*

3.3 Polygonal Meshes

It is very common in computer graphics to approximate a shape's surface with a polygonal mesh (see Definition 3.4). Consequently, to generate and modify meshes procedurally, mesh data structures need to be designed appropriately. Characteristic and commonly used polygonal meshes are *triangle meshes* and *quadrangle (quad) meshes*, which only exist of triangles, respectively quadrangles, as faces. More general *boundary representations* or short *B-reps* contain faces of arbitrary degree. These three types of meshes are explained in short in the following. For more detailed information please consult the books by Mäntylä [Män88], de Berg et al. [dBvKOS00], or Botsch et al. [BKP*10].

This section will then proceed with an overview on mesh data structures for storing and editing that are used and maintained in procedural generation and modification of meshes (see Section 3.3.1). This section concludes with a discussion on the operators needed for mesh navigation as well as mesh generation and editing. These operators are necessary to reach a procedural mesh description (see Section 3.3.2).

Triangle Meshes.

A triangle is the smallest two-dimensional polytope. Due to this, it is possible to approximate any three-dimensional surface up to arbitrary precision through triangles. Consequently, triangles are the basic primitive for describing surfaces with polygonal meshes and modern graphic card hardware is optimized for rendering triangles.

Interesting properties about triangle meshes are that there are in average three times as many edges as vertices and in average twice as many faces as vertices. For a manifold triangle mesh $M = (V,E,F)$, let $v = |V|$, $e = |E|$, $f = |F|$ be the number of vertices, edges, and faces respectively. In such a mesh, $3f = 2e$ by counting the edges of faces. By inserting this in the Euler-Poincaré formula (see Definition 3.9), we reach $f \approx 2v$ and $e \approx 3v$. Consequently, the amount of vertices and faces together approximate the number of edges in a triangle mesh.

Another interesting property is that the average valence of vertices is 6. The average valence can be computed by $2e/v$, which is approximately 6 for large v.

Quadrangle (Quad) Meshes.

Quadrangles, on the contrary, do not share the topological features of triangles. Four points in general position do not necessarily need to be in one plane or in convex position. However, in a sense it feels much more natural to express a shape's surface with near quadratic patches. To generate a quad mesh of a surface it is important to know that each point on a surface exhibits two orthogonal tangent directions of principal curvature [Eul67]. One characterizes the rate of maximum bending and the other the rate of minimum bending. These directions can be used to represent a surface by quadrangular patches, hence by a quad mesh.

Quad meshes are also often used for free-form architecture that features glass façades. These façades should consist of planar quadrangular window patches. Planarity is an important factor here because bent glass panes are very expensive to fabricate.

The topic of quadrangulation of meshes received much attention recently. Tarini et al. [TPC*10], for example, presented a simplification-based approach. They first convert a triangle mesh into a quad mesh by pairing triangles and simplify it afterwards using specific mesh operations.

Boundary Representations (B-reps).

Triangle meshes and quad meshes are special variants of the more general class of meshes: the so-called boundary representations, or short B-reps. B-reps allow faces of arbitrary degree and those faces do not need to be regular nor convex. Furthermore, it is a good idea to allow edge rings within faces. Through edge rings (or loops) additional borders are added to a face. A representation with edge rings can be transferred to one without by connecting these two borders with a pair of edges.

3.3.1 Mesh Data Structures

To efficiently store meshes it is not necessary that all information of a mesh (see Definition 3.4) is stored explicitly. Depending on the application, a trade-off between required space to store the mesh information and time to query information is necessary. In this section different data structures to save and operate on mesh data are explored. Data structures for saving are concise and store information only implicitly; therefore, they need to be converted to other more explicit data structures for mesh editing. These data structures for editing allow to define a set of operators for procedural generation and modification of meshes (see Section 3.3.2).

Mesh Data Structures for File Storage. There exist many different file formats for storing mesh data. In this part I examine the most common variants. A primitive solution to describe triangle meshes was taken within the .stl file format. Each line in the file features nine floating-point values – three coordinates for three vertices – and therefore represents one triangle. While this file format is intuitively simple, it has one major drawback: there is a lot of redundant information. Vertices that are shared by triangles appear multiple times and adjacency information can only be retrieved by comparing floating-point values. To prevent redundancies within the vertices, a vertex list with unique entries is necessary. Vertices are then only referred to by their unique identifier.

Another way to store triangle meshes efficiently is via so-called *triangle strips* that interpret a list of vertex indices in a special way. Starting from an initial triangle, which is encoded by the first three vertices, each further vertex generates a new triangle together with the two preceding vertices. This is very efficient because each vertex (except the first two) generates a new triangle. However, it is necessary to resort to creating degenerated triangles by using vertices multiple times to express arbitrary triangle meshes this way (see Figure 3.4). By this means, it is possible to gap holes and continue the triangle strip at another position. Even though this creates additional triangles with no surface area that no not contribute to the surface of the mesh, this method is ideal for rendering purposes and is used in OpenGL and DirectX because it can be processed very fast and efficiently.

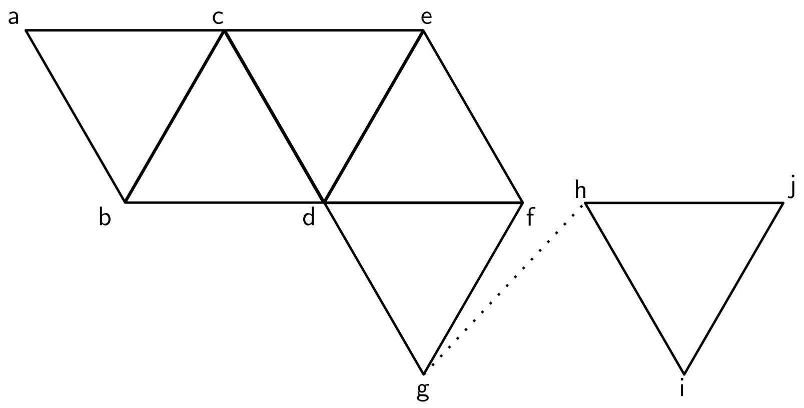


Fig. 3.4 These triangles are represented as a triangle strip by the vertex list “abcdeffdgghhij”. Each subsequent triplet of vertices represents a triangle. The triangles abc, bcd, cde, and def are represented by the list “abcdef”. To traverse from triangle def to triangle fdg, the vertex f needs to be used twice to insert the degenerated triangles eff and ffd. Afterwards, the triangle fdg can be created. The same method can be used to relocate the triangle strip to the triangle hij by using the vertices g and h twice.

A way to describe arbitrary meshes are the so-called *indexed face sets* (see Figure 3.5). Outgoing from a list of vertices, faces are defined by a list of vertex indices. Different file formats use different delimiters to signal the ending (or beginning) of a new face. File formats that use indexed face sets are for example the Wavefront OBJ file format (.obj), the Object File Format (.off), or the Virtual Reality Modeling Language (VRML) file format (.wrl). Furthermore, this representation is widely accepted by modern graphics hardware, therefore no conversion of the data is necessary before rendering, which makes this representation ideal to store render data.

v_0	(0,0,0)
v_1	(1,0,0)
v_2	(1,1,0)
v_3	(0,1,0)
v_4	(0,0,1)
v_5	(1,0,1)
v_6	(1,1,1)
v_7	(0,1,1)

f_0	v_0, v_3, v_2	f_6	v_2, v_7, v_6
f_1	v_0, v_2, v_1	f_7	v_2, v_3, v_7
f_2	v_0, v_5, v_4	f_8	v_3, v_4, v_7
f_3	v_0, v_1, v_5	f_9	v_3, v_0, v_4
f_4	v_0, v_6, v_5	f_{10}	v_4, v_5, v_7
f_5	v_1, v_2, v_6	f_{11}	v_5, v_6, v_7

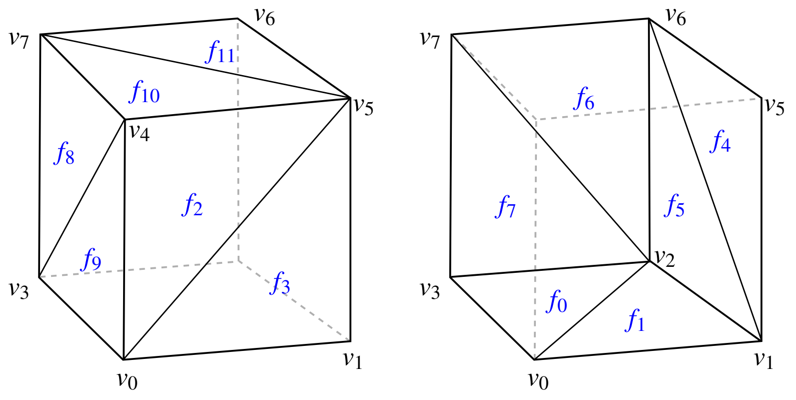


Fig. 3.5 In an indexed face set data structure vertices are stored in a list without redundant entries. Faces are represented by a list of vertex indices.

Mesh Data Structures for Editing. Meshes that are saved to files store adjacency information only implicitly to save storage space. However, should a mesh be subject to editing then this adjacency information is important. The adjacency information can be extracted in $O(n)$ time from such a representation, but for editing purposes this takes too much time. Therefore, a trade-off between time and space, which is a conversion to another representation, namely a *halfedge* representation, is necessary. In contrast to indexed face sets, which are ideally for storing mesh data, the halfedge representation is ideally suited for interactive modeling, where changes to the mesh are done locally. The ability to navigate on the mesh freely enables dynamically changing the mesh easily, thus the mesh can be described through procedural modeling (see Section 3.3.2). This representation allows to query the adjacency information in $O(1)$, but also requires a lot of space, which can be a problem for very large meshes.

The halfedge representation is the successor of winged-edge meshes by Baumgart [Bau75] and is the one of the most flexible representations in terms of editing. The entire vertex and face information is stored and, additionally, directed halfedges are stored for each face to represent the orientation of that edge in that specific face. This amounts to two halfedges per edge, so twice the amount of space is needed for storing the edge information.

Halfedges can be used to encode adjacency information of a mesh. Figure 3.6 provides an overview on the entire adjacency information that can be stored in halfedges. Aside from the information stored in vertices and faces, it is not always necessary to store the whole information that is shown in Figure 3.6 in the halfedges. Triangles meshes, for example, can be efficiently represented through a list of halfedges. In this list, triplets of halfedges always encode one triangle in counter-clockwise direction. Consequently, for a given halfedge with index i the index of the triangle can be computed by $\lfloor i/3 \rfloor$, and the index of the halfedge within the triangle can be computed by $i \pmod 3$. This way, the information to navigate around a triangle is stored implicitly. To enable navigation over the whole mesh each halfedge needs to store the infor-

1. each vertex references one outgoing halfedge
2. each face references one half edge of its boundary
3. each halfedge references:
   a. the vertex it is pointing to
   b. the adjacent face
   c. the next halfedge (of the corresponding boundary)
   d. the opposite halfedge (mate)
   e. optionally the previous halfedge

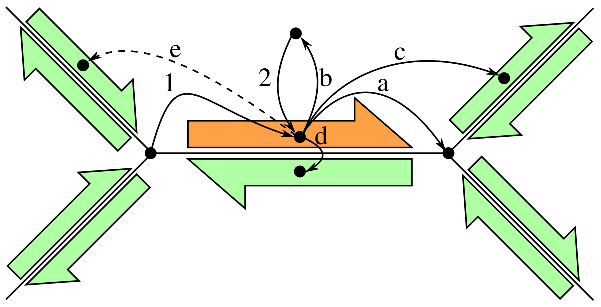


Fig. 3.6 Illustration of the halfedge data structure and the complete information that can be stored within halfedges.

mation about their edge mate explicitly. Additionally, information of the vertex the halfedge is pointing to is necessary for mesh editing purposes.

In case of the more general B-reps, halfedges need to be stored differently because faces do not necessarily have the same amount of edges. In B-reps the edge mate information is saved implicitly, which means that the list of halfedges consists of pairs of halfedges that are each other's mates. For a halfedge with index i the mate has index $i+1$ if i is even and $i-1$ otherwise. To enable navigation around the mesh, the index of the next halfedge in counter-clockwise orientation is stored explicitly for each halfedge. Again it is necessary to store a reference to a vertex explicitly in the halfedges.

3.3.2 Mesh Editing and Navigation Operations

Meshes are usually stored in files as indexed face sets, but when meshes are to be changed interactively the underlying mesh information (once loaded into the software) is usually converted to a halfedge representation. The advantages of the halfedge representations are the ability to navigate the mesh freely, which enables changing the mesh dynamically. For saving the mesh, which has been edited this way, a conversion to another format is necessary because the halfedge data structure is not space efficient. The halfedge representation is the standard representation for editing polygonal meshes. This section focuses on the available operations to navigate and edit a mesh, which is represented through halfedges. Through the combination of navigation and edit operations, procedural generation and modification of polygonal meshes is possible.

Mesh Navigation Operators for Procedural Modeling. Halfedges can be seen as *mesh iterators* that can be used for traversing meshes based on the information they store. The following operators are used for navigation on any polygonal mesh and operate in constant time:

faceCCW and **faceCW**
Traverse the edge cycle of a face's boundary in either counter-clockwise (CCW) or clockwise (CW) direction.

vertexCCW and **vertexCW**
Traverse the edge cycle around a vertex in either counter-clockwise (CCW) or clockwise (CW) direction.

mate
For a given halfedge get the opposite halfedge of the neighboring face.

Outgoing from an arbitrary halfedge on a valid mesh each other halfedge (on the same connected component) can be reached through application of these five operations. Any path between two vertices on the same connected component can be realized through application of these navigation operations.

Mesh Editing Operators for Procedural Modeling. A mesh can be seen as a graph of vertices that are connected by edges, which again enclose faces. With a polygonal mesh given, one might desire to edit it by adding or deleting vertices, edges, or faces. This way, the original mesh is transformed into a new mesh.

Basic editing and generation operations for triangle meshes of constant genus have been introduced by Hoppe [Hop96] in the context of reversible mesh simplification. Additional to a *make triangle* operation that creates an initial triangle, he introduced a pair of mutually inverse operations.

- The *vertex split* operation takes a vertex and divides it into two by introducing a new edge connecting the two vertices. By introducing this new edge, two existing edges, which are incident to the split vertex, are split too. These edges span – together with their original counterpart and the new edge connecting the split vertices – two new faces as illustrated in Figure 3.7 from left to right.
- The *edge collapse* operation takes an edge and deletes it by merging the connected vertices into one single vertex. This operation consequently deletes two faces and merges two pairs of existing edges as illustrated in Figure 3.7 from right to left.

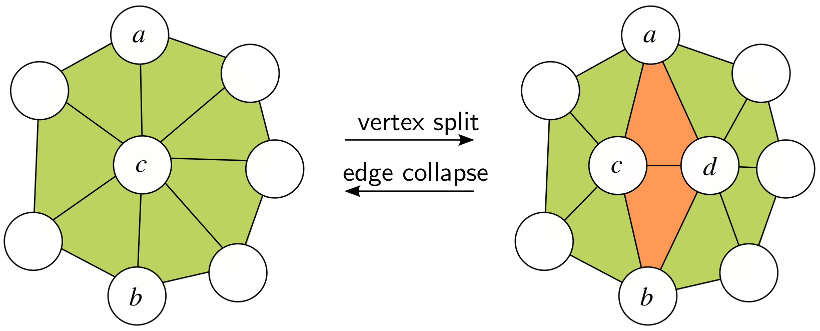


Fig. 3.7 Illustration of the mutually inverse operations vertex split and edge collapse. The vertex split operation (left to right) splits the vertex c into two generating the new vertex d. This way, three new edges and two faces (in orange) are created. The inverse, the edge collapse operation, merges two vertices into one by deleting the vertex d together with the two orange faces and three edges.

The three operations *make triangle*, *vertex split*, and *edge collapse* can be generalized to describe the generation of an arbitrary polygonal mesh. This complete set of operations, the so-called *Euler operators*, can be used to describe a mesh procedurally as discussed in the work of Havemann [Hav05]. During the process of mesh editing with Euler operators the mesh does not necessarily be valid all the time. However, at each step the Euler-Poincaré formula (see Definition 3.9) holds. These operators are the foundation of modeling with the scripting language GML, which in introduced in Section 4.1.

There are two groups of Euler operators: the *make group* and the *kill group*, whereas for each operation in the make group the corresponding inverse operation is in the opposite kill group and vice versa. Figure 3.8 illustrates all Euler operators.

First, three operators are introduced to create objects of genus zero:

makeVFS ($\leftrightarrow$ killVFS): Make Vertex, Face, and Shell
This operation creates the connected component (shell) together with one face and one vertex for initializing the modeling process. At this step the shell and the face are infinitely small. As the model will be generated with further steps, this initial face will always be the face that closes the mesh. The inverse deletes a shell together with the last vertex and face.

makeEV ($\leftrightarrow$ killEV): Make Edge and Vertex
Based on one vertex this operation generates another vertex and the connection between these two. Hence, this operation either creates a *dangling vertex* with valence one, or it splits the original vertex, together with its incident edges, into two. The inverse operation removes an edge and the two end points become one.

makeEF ($\leftrightarrow$ killEF): Make Edge and Face
This operation is dual to the makeEV operation as it splits a face into two introducing a new edge and a new face. Therefore, it is a *face split*. Either it creates a face loop within an existing face by taking the same vertex as start and end of the edge, or it splits a face in two pieces by connecting two different vertices of that face. The inverse is also called *face join*.

These three pairs of matching operators can be used to create any object, which is topological equivalent to a sphere. In a forward modeling process one would always start with an initial makeVFS to create the shell together with first vertex and a face. Modeling continues by application of makeEV and makeEF operators. A sequence of makeEV operators is used to create all vertices with the corresponding edges. The faces are then finished by inserting the last edge that is missing for each face with the makeEF operator.

In comparison to the mesh editing operations on triangle meshes, the Euler operators operate on a lower level. A vertex-split operation, for example, which introduces one new vertex, two new faces, and three new edges, amounts to one makeEV and two makeEF operations. Therefore, on triangle meshes the Euler operators are equivalent to the vertex-split and edge-collapse operations. The latter, however, do not provide facilities to introduce genus changes to the mesh. To grant the flexibility of topological changes, such as making holes or connecting two components, the following two pairs of operators, which are based on the concept of edge rings within faces, are added to the list of Euler operators.

makeEkillR ($\leftrightarrow$ killEmakeR): Make Edge, Kill Ring
This operator is used to connect one isolated edge ring with the face on which the ring is positioned. The newly inserted edge connects two points of the same face, one on the outer boundary and one of the ring. Through this connection, the ring vanishes. In case of forward modeling, the inverse killEmakeR operator is used to create a ring within a face.

makeFkillRH ($\leftrightarrow$ killFmakeRH): Make Face, Kill Ring and Hole
This last operation is used to turn a ring into a face of its own. Again in case of forward modeling, the inverse killFmakeRH operator is used and is easier to illustrate. With the inverse operator, a face is turned into a ring of another face. This can be used to change the genus of a mesh. On the one hand, it can be used to connect two shells with two coplanar faces touching each other, and, on the other hand, it can be used to create topological holes when two faces of the same shell are coplanar.

The operators makeEkillR and makeFkillRH can be used to convert faces to rings and vice versa.

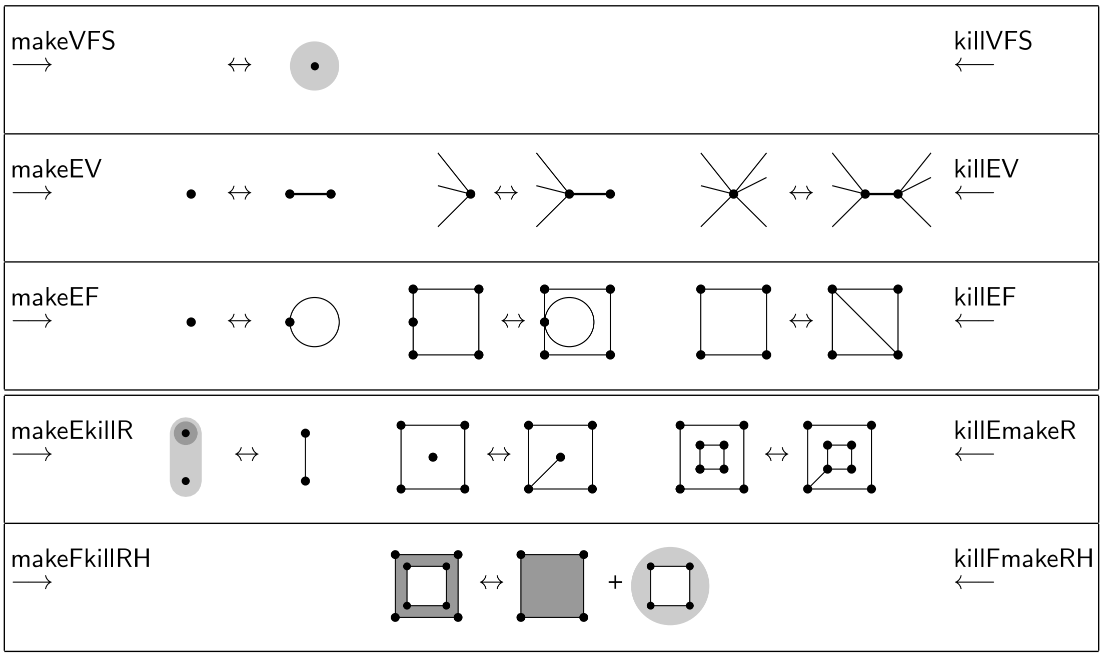


Fig. 3.8 Illustration of the Euler operators in two-dimensions. The make operators on the left correspond to their counterpart kill operators on the right. Each row shows different situations for applying the operator. The make operators are applied from left to right, and the kill operators from right to left. With the first three operators, any genus zero shapes can be generated. The last two operators are concerned with rings and building higher-genus objects. (image source: Havemann [Hav05])

It is important to notice that the Euler-Poincaré formula from Definition 3.9 does never fails after application of one of the mentioned Euler operators. Figure 3.9 shows the contribution of each operator to the values used in the formula.

Euler Operator	v	e	f	s	r	h
makeVFS	+1		+1	+1		
makeEV	+1	+1				
makeEF		+1	+1			
makeEkillR		+1			-1	
makeFkillRH			+1		-1	-1

Fig. 3.9 Table showing the contribution of each Euler operator of the make group to the values used in the Euler-Poincaré formula (see Definition 3.9). The contributions of the operators from the kill group are the negative values of the corresponding counterpart from the make group. No operator would cause the Euler-Poincaré formula to fail.

Euler operators form a complete set of modeling operations, which can be utilized by procedural modeling techniques, for manifold solids. More precisely, from an initial topologically valid mesh each other topologically valid mesh can be constructed by a finite sequence of Euler operators.

3.4 Subdivision Surfaces

Describing highly tessellated curved surfaces procedurally through the aforementioned Euler Operators (see Section 3.3.2) is inefficient due to the high amount of faces needed to generate a smooth surface. Consequently, we want to find a method to allow a more efficient procedural description of curved surfaces.

Subdivision surfaces bridge the gap between polygonal models and spline surfaces. They approximate a free-form surface through a infinite process of subdividing faces of any two-manifold polygonal mesh into smaller pieces. A smooth surface can so be represented by a coarse polygonal mesh. With the appropriate subdivision rules a cube can be refined to become a sphere with infinite subdivision steps (see Figure 3.10). Subdivision surfaces are an important modeling tool within the scripting language GML, which is introduced in Section 4.1.

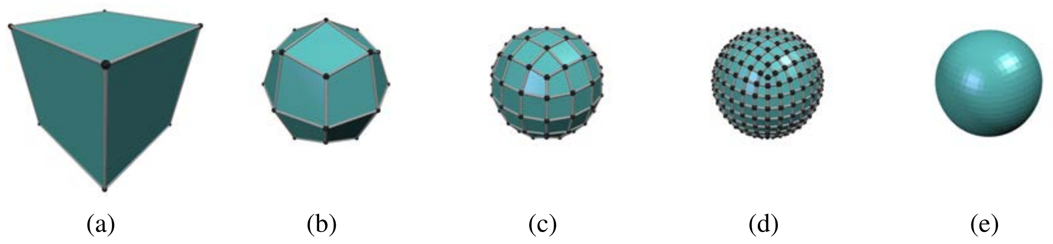


Fig. 3.10 Catmull-Clark subdivision steps applied to a base mesh formed like a cube (a). The subdivision steps up until subdivision depth three are seen from (b) to (d). After infinite subdivision steps the limit shape is reached. In this example the limit shape is a sphere (e).

Terminology of Subdivision Surfaces. A *subdivision surface* is defined by a *base mesh* M^0 with its vertices being called the *control points* cp^0 and a set of *subdivision rules*. By applying the subdivision rules to a mesh M^n the next iteration of the mesh M^{n+1} and control points cp^{n+1} are reached. The index n is called the *subdivision depth*.

Mathematically, the approximated smooth surface is the limit of a recursive subdivision process of each face into smaller faces that better approximate a surface. The smooth surface is therefore called *limit surface* $L(M_0) = \lim_{n\to\infty} M^n$. The limit surface is no parametric surface, but can be evaluated directly for most subdivision schemes. Thus, the recursive refinement steps become unnecessary.

To subdivide a face of a mesh, an affine combination of neighboring elements of that face are used to calculate the positions of the new vertices. Subdivision methods can either be *interpolating* or *approximating*. With interpolating methods, control points are placed on the limit surface. Thus, the original positions of the vertices are required to stay the same in the whole subdivision process. In contrast, with approximating methods control points are typically not placed on the limit surface. The position of control points can therefore vary from iteration to iteration.

Independent of the subdivision scheme, subdivision surfaces are either refined *uniformly* or *adaptively*. When a base mesh is subdivided sufficiently often it becomes a good approximation of the limit surface. For a uniform refinement, subdivision rules are applied for a number of iterations to all faces in each iteration; so there is a overall constant subdivision depth. This, however, leads to an exponential memory consumption and to subdivisions on faces where a refinement produces almost no accuracy gain. Adaptive refinements, on the other side, vary the subdivision depth, but ensure a constant accuracy. Parts with high curvature will be subdivided more often, whereas flat parts will hardly be refined. To ensure a coherent result, it is necessary to use limit points for the connection of mesh parts with different subdivision depths.

Properties of Subdivision Surfaces. Subdivision surfaces have many positive features. Arbitrary topology can be described and processed with subdivision surfaces. Due to the recursive refinement, level of detail can be obtained very easily by setting the maximum subdivision depth. The recursive definition also leads to an easy and structured implementation.

However, problems with the curvature occur at irregular vertices and faces. Irregular vertices are points where the mesh is not regular. In quad meshes, an irregular vertex is a vertex with valence unequal to four. Depending on the subdivision algorithm curvature is hard to control at these points (see Figure 3.11). Irregular faces are faces that are not suited for the subdivision algorithm. For a subdivision procedure that is defined on quad meshes, faces of other degree than four are considered irregular. Sometimes, faces are also considered irregular if they contain an irregular vertex. There are several papers dedicated on how to control and handle irregular vertices and faces. For example, Augsdörfer et al. [ADS06] analyzed artifacts of subdivision schemes to tune them for the best possible behavior near irregular vertices with respect to curvature variation.

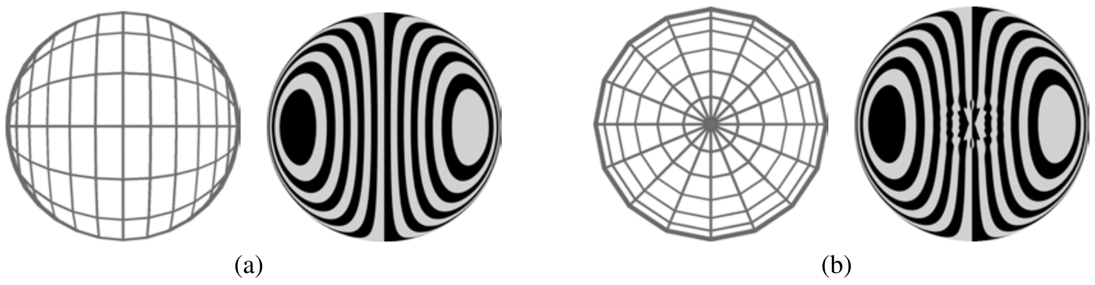


Fig. 3.11 Influence of irregular points on the curvature. The images show on the left side the control points and on the left side the curvature is visualized by reflecting a texture of parallel black lines on a white background. In regions with no irregular points (a) the curvature is smooth, but around an irregular point (b) the curvature is disturbed, which leads to artifacts in reflections. (image courtesy of Andreas Riffnaller-Schiefer)

Catmull-Clark Subdivison Procedure. The Catmull-Clark subdivision algorithm [CC78] by Catmull and Clark is an approximating scheme as a generalization of bi-cubic uniform B-spline surfaces. This subdivision algorithm is one of the most commonly used and is also realized within the implementation of subdivision surfaces in the GML (see Section 4.1.2).

Each control point of a mesh M^{i+1} can be associated with a face, edge or vertex of the mesh M^i of the previous iteration; these points are therefore called *face points*, *edge points*, or *vertex points*. These points are calculated and then form the control points of the mesh for the next iteration.

For a mesh M^i at the i^{th} iteration with control points P^i a subdivision step proceeds as follows:

1. For each face F_j a *face point* f_j^{i+1} is created. This point is the average of all control points P_k^i that belong to that face – the centroid. So

$$f_j^{i+1} = \frac{1}{n} \sum_{k=1}^{n} P_k^i, \tag{3.1}$$

   where n is the amount of vertices P_k^i of that face

2. For each edge E_j an *edge point* e_j^{i+1} is created by taking the average of both end points (P_a^i and P_b^i) and the newly created face points of the neighboring faces F_k and F_l.

$$e_j^{i+1} = \frac{P_a^i + P_b^i + F_k^{i+1} + F_l^{i+1}}{4}. \tag{3.2}$$

3. For each control point P_j^i a *vertex point* v_j^{i+1} is created. To compute the position of this vertex point all neighboring face points and edge midpoints from adjacent edges and faces are taken into account. Let R be the average of all n face points created from faces incident to P_j^i and let, furthermore, Q be the average of all n edge midpoints of edges incident to P_j^i, then

$$v_j^{i+1} = \frac{R + 2 \cdot Q + (n-3) \cdot P_j^i}{n}; \tag{3.3}$$

   which is the barycenter of P_j^i, Q and F with respective weights $(n-3)$, 2 and 1.

4. Edges for the new mesh are formed by connecting each face point with all edge points that belong to the edges that are incident to the face the face point was derived from. Furthermore, for each created vertex point v^{i+1} from a control point P^i, all edge points of edges incident to the old control point P^i are connected to v^{i+1}.

5. The faces of the subdivided mesh M^{i+1} are enclosed by the edges described by all the face points, edge points and vertex points from the new control points P^{i+1}.

Independent from the start mesh M^i, M^{i+1} will always be a quad mesh. Extraordinary points emerge from faces that were no quads to begin with. Due to the fact that a quad mesh is created, the amount of extraordinary vertices remains constant over successive refinements. Repeated subdivision yields a smoother mesh. At the limit surface C^1 continuity can be guaranteed at extraordinary vertices and C^2 continuity everywhere else. In the example of Figure 3.10 the above described subdivision steps were used.

Stam [Sta98] presented a method where a Catmull-Clark surface can be evaluated without the refinement process. He reformulated the subdivision process into a matrix exponential problem. Such problems can be solved directly by matrix diagonalization methods.

Creases for Catmull-Clark Surfaces. Catmull-Clark subdivision always creates smooth continuous surfaces. To add more versatility to subdivision surfaces, DeRose et al. [DKT98] from Pixar Animation Studios introduced sharpness and creases to Catmull-Clark surfaces (see

Figure 3.12). To handle these new features new subdivision rules for sharp edges and vertices need to be defined. An integer *sharpness value n* is assigned to each edge. This value defines whether the edge is processed as a *smooth* edge ($n = 0$) or a *sharp*, respectively *crease* edge ($n > 0$). Smooth edges are processed with the ordinary subdivision rules, but for crease edges modified rules are used. Furthermore, each time a crease edge is processed its sharpness value n is reduced by one for the next iteration.

For crease edges the calculation for edge points is changed to the midpoint of the edge. So for an edge point e_j^{i+1} is obtained by

$$e_j^{i+1} = \frac{P_a^i + P_b^i}{2}, \tag{3.4}$$

where P_a^i and P_b^i are the respective endpoints of the edge.

The calculation of vertex points are modified dependent on how much crease edges are incident to a specific vertex. Vertices that have exactly one incident sharp edge are called *dart vertices*, vertices with exactly two incident sharp edges are called *crease vertices*, and all other vertices with more than two incident sharp edges are called *corner vertices*. Those three vertex types are processed differently:

Dart Vertex

Dart vertices are processed like smooth vertices with the formula provided in Equation 3.3.

Crease Vertex

For the two sharp edges from P_k^i to P_j^i and from P_j^i to P_l^i the new vertex point v_j^{i+1} is positioned at

$$v_j^{i+1} = \frac{P_k^i + 6 \cdot P_j^i + P_l^i}{8}. \tag{3.5}$$

Corner Vertex

Corner vertices do not move in the subdivision process, so the vertex point v_j^{i+1} for the next iteration is equal to the control point P_j^i of the current iteration, so

$$v_j^{i+1} = P_j^i \tag{3.6}$$

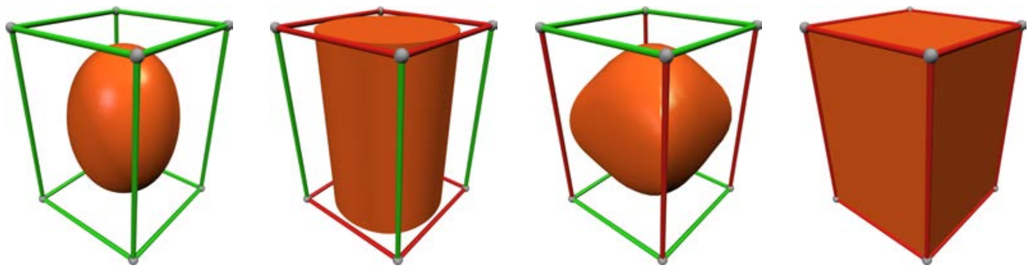


Fig. 3.12 Influence of sharp edges in a control point mesh formed like a cuboid on the Catmull-Clark subdivision surface. Green edges represent smooth edges and red edges represent edges with sharpness infinity.

An Operator Interface for Subdivision Surfaces. Subdivision procedures refine manifold polygonal meshes to reach smooth surfaces. Therefore, the Euler operators (see Section 3.3.2) can be utilized to create any base mesh to which the subdivision algorithms are applied to. Additional operators are needed that assign sharpness values to edges and vertices to procedurally facilitate creases in subdivision surfaces. These additional operators take a halfedge and a sharpness value as input and assigns the latter to the corresponding edge, respectively vertex. Subdivision surfaces are used within combined B-reps in the GML (see Section 4.1.2)

3.5 Convex Polyhedra

So far only surface representations have been discussed. Within these representations it is important that the corresponding mesh is always closed and valid. This can be ensured through appropriate use of the Euler Operators (see Section 3.3.2). On the contrary, in a volumetric representation of a shape there is always a well-defined inner and outer part and the surface of the shape is the boundary between them. Consequently, the surface is always closed and needs no maintenance. In the focus of this section is an accurate volumetric shape representation based on convex polyhedra, which, furthermore, provides a higher-level operator set than Euler Operators for the procedural creation of shapes.

A polytope is a geometric object with straight boundary elements that can be defined in arbitrary dimensions. In two dimensions a polytope is called polygon and in three dimensions it is called polyhedron. A *convex polyhedron* is a special case of a polyhedron, where all points on its surface are in convex position. Platonic solids (see Figure 3.13) are special convex polyhedra, which have congruent faces of regular polygons and the same number of faces meeting at each vertex.

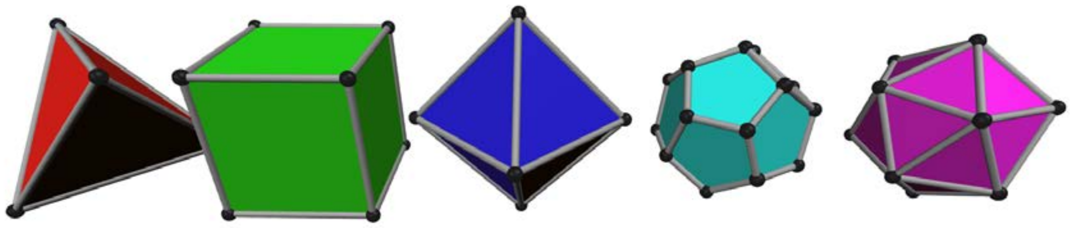


Fig. 3.13 The five Platonic solids are special convex polyhedra. All faces are congruent regular polygons and the same number of faces meets at each vertex. From left to right there are the tetrahedron (four triangles), the cube (six quadrangles), the octahedron (eight triangles), the dodecahedron (twelve pentagons), and the icosahedron (twenty triangles). These five solids are the only shapes that fulfill the mentioned properties.

Representing convex polyhedra through meshes is feasible when accuracy does not play an important role. Calculations on a mesh represented by vertices with floating-point coordinates always involve errors. Due to the inexact representation of floating-point values, it is, for example, not possible to guarantee that three lines (or four planes) that should intersect in one common point will do so when represented through floating-point coordinates. This also complicates testing on which side of a given plane a given point lies because, in general, vertices of faces with higher degree practically never lie in the same plane. So for an accurate representation of convex polyhedra another description is necessary.

An important observation is that a convex polyhedron can be described by intersections of half spaces.

Definition 3.11 *A* ***half space*** *H is described by a tuple $(\vec{n}, d)$, where $\vec{n}$ is the normal vector of a plane and d is its offset. Thus a half space is the set of points p that satisfy the inequality $\langle \vec{n}, p \rangle < 0$. This set of points is said to be in the* ***interior*** *of H.*

The complement $\overline{H}$ of a half space H is simply obtained by reversing the normal vector, so for $H = (\vec{n}, d)$, $\overline{H} = (-\vec{n}, d)$. The points p that satisfy the equality $\langle \vec{n}, p \rangle = 0$ lie on the boundary of the half space and define an oriented plane in space, which in turn can be used to define a half space. A convex polyhedron can now be defined as an intersection of half spaces.

Definition 3.12 *A* ***convex polyhedron*** *P can be defined by a set of half spaces $\{H_1, H_2, \dots H_n\}$ and is the set of all points that lie in the interior of all these half spaces.*

An intersection of half spaces can be empty which yields in an *empty* polyhedron. Since half spaces are convex, the intersection of half spaces remains convex too.

A Fast and Exact Representation for Convex Polyhedra. To provide a shape representation that can be processed and refined further without loss of accuracy, operations need to be designed in an appropriate way. In the work of Krispel et al. [KUF14] they present a fast and exact plane-based representation for polygonal meshes. Instead of representing vertices with three floating-point values, they follow the approach from Sugihara et al. [SI90], where a vertex, which is the result of intersecting planes, can be exactly represented by the set of three planes that intersect at that point. The conversion to floating-point values should only be done when no further processing is necessary, for example for rendering purposes, and should not be used in any calculations.

To avoid inaccuracies in plane-based mesh modification operations, they restrict the resolution of the coefficients of planes, which in turn limits the space of available planes. In their work they present a trade-off between the available resolution of planes and fast calculations based on them. This allows the resolution to be adapted and optimized to guarantee fast calculations for any target platform. Further details are available in their work [KUF14] and in the Ph.D. thesis of Krispel [Kriar].

An Operator Interface for Convex Polyhedra. A natural operation for this kind of representation is the trimming of a convex polyhedron with a plane. By adding one additional plane to a polyhedron the parts of the polyhedron inside the half space corresponding to this plane remains and the part outside is discarded. So adding a plane to a polyhedron means cutting off a piece of this polyhedron. This add-plane operation is the basic operator for creating convex polyhedra procedurally. Any convex polyhedron can be expressed by a list of applied add-plane operations. Of course, an inverse remove-plane operation is also necessary to provide a complete set of operations.

Essentially, the add-plane is enough to describe any convex polyhedron, but for convenience further operators can be imagined. A split operation amounts to an intersection of a polyhedron P with a half space H and its complement $\overline{H}$, which are both defined by the same plane. Therefore, both results of the split operation are non-overlapping and fill the same volume as the original polyhedron P. The two resulting polyhedra are formed be the half spaces of P, as well as H and $\overline{H}$ respectively. To realize such a split operation efficiently, a duplicate-polyhedron and flip-halfspace operator can be introduced. Moreover, operators for placement of half spaces can be envisioned that ease the construction process. Essential operators of the interface by Krispel [Kriar] for modeling with convex polyhedra in the GML are explained in Section 4.1.2.

A Suitable Procedural Method for Convex Polyhedra: Split Grammars. Split grammars are a suitable application for these robust operations (see Figure 3.14). This way models in split grammars are described by a set of non-overlapping convex polyhedra achieved through the split operations. Each polyhedron is a single closed object and their touching faces are superfluous geometry when the model is exported. To generate a surface mesh of the generated model, the outer surface of a set of touching polyhedra can be extracted, which removes unnecessary geometry on the inside. Section 4.2 focuses on split grammars achieved through convex polyhedra. For a detailed explanation on how to prepare convex polyhedra as a modeling primitive and how to use them for split grammars please refer to the work of Krispel [Kriar].

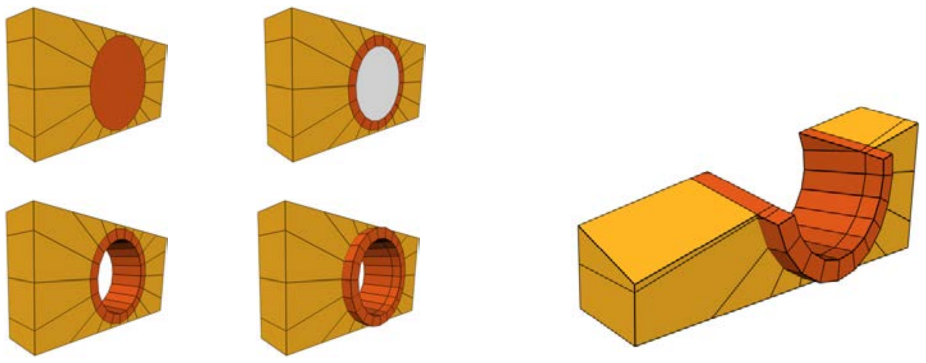


Fig. 3.14 The plane-based description of convex polyhedra is robust to the application of continuous plane splits and is consequently suited for the application of split grammars. In this example, detail (hole with border) is generated through several splits and displacing of boundary planes (extrusion). Adding an additional splitting plane is always always possible as indicated on the right.

3.6 Scene Graphs

So far only low-level shape representations have been reviewed and an operator interface for them was presented. Now the focus shifts from single objects to ways to set these objects into context. The first structuring method that is discussed in this chapter is a scene graph, which allows for procedural placement, animation and interconnection of three-dimensional data. A detailed introduction to scene graphs and their use within computer games can be found in the book of Eberly [Ebe06].

Scene graphs are used for spatial arrangement of data and are realized as directed acyclic graphs. Like other graphs, it is composed of a collection of nodes and directed connections between them. In scene graphs these connections define a “parent-child” relationship. All effects applied to a node in a scene graph influence all children of this node – the sub-graph adjacent to this node – as well. Nodes are generally used to carry all kinds of information from geometric data to information about materials or transformations. The relationships between nodes are often used for grouping of several objects, so that all objects behave as one. In this case a transformation would affect all nodes, not just one. Another use case for this relation is to map dependencies (see Figure 3.15). A vase of flowers is standing on a table. The vase itself can be moved on the table, but moving the table moves the vase too. Scene graphs have been included in the scripting language GML (see Section 4.1.2) and have been made accessible for interactive modeling in a separate tool (see Section 4.4.4).

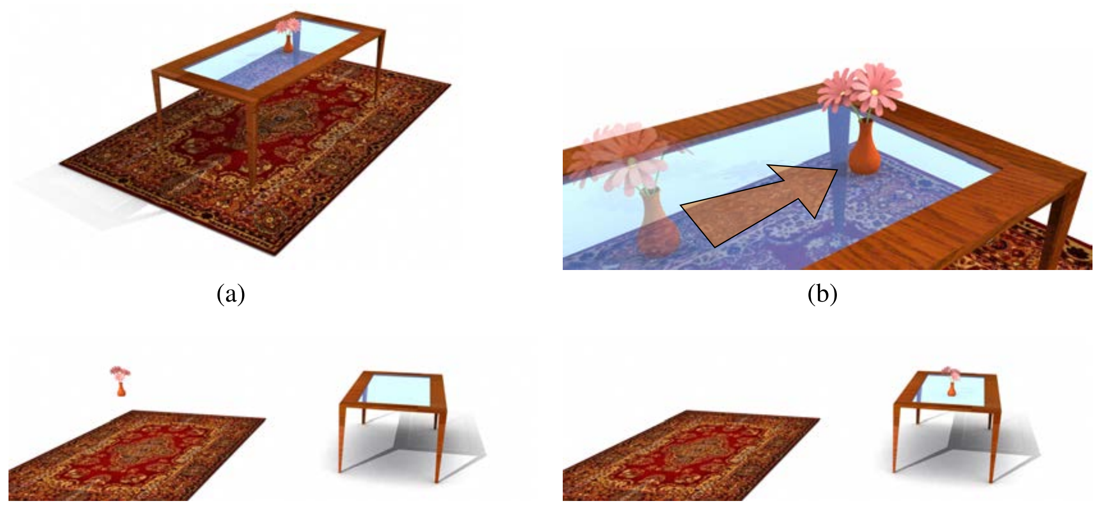

(a) (b) (c) (d)

Fig. 3.15 Without scene graphs all objects can be moved independently. Let us take a setup with a flower vase on a table (a). The flower vase can be moved on top of the table (b) without influencing the position of the table, and the table can be repositioned (c) without affecting the vase. With scene graph systems a hierarchical relation between objects can be established. The table enters a parent-child relationship with the vase. Consequently, moving the table moves the flower vase too (d), while the vase can still be moved independently (b).

A Simple Operator Interface for Scene Graphs. Operators for building a graph are necessary to create scene graphs procedurally. Such operators would include operators such as create-node and add-child (and, of course, the inverse remove-node and delete-child operators). With these operators any graph can be constructed. To gain the functionality of a scene graph, it is necessary to assign certain properties to specific nodes. To achieve the basic functionality of a scene graph, an add-transformation operator and an add-geometry operator would be sufficient. To provide a complete set of operators, an inverse reset-node operator is necessary as well. In Section 4.1.2 the realization of an operator set for procedural modeling of scene graphs by Hecher [Hec12] in GML is presented. These procedural scene graphs have been used to create procedurally animated environments (see Section 6.4 and 6.5).

3.6.1 Applications of Scene Graphs

The concept of a scene graph appears in many different fields of application, but it is rarely called a scene graph. Graphs are often used internally in software to define, for example, groups, a hierarchy, or the visibility of objects. All of these applications can be interpreted as a usage of a scene graph. The main fields of application of scene graphs are two-dimensional graphic design tools and three-dimensional applications, such as simulations or games.

Grouping in Two-Dimensional Graphic Design Tools.

Graphics tools, in which scene graphs are utilized, include vector graphic tools, computer-aided design tools, as well as general graphics programs for rasterized images. Leaf nodes of scene graphs represent atomic units in these systems. These atomic units are for example lines, circular arches, splines, or images. Scene graphs are utilized to group these objects and transform and edit them as they were one single object. The layer concept is very similar to this grouping approach. Layers act as sheets in which objects can be placed. These sheets can then be triggered visible, invisible, or locked. Essentially this is no different from the before mentioned group concept, it is just utilized in another way. So groups and layers are all nodes of the same scene graph.

Relations in Three-Dimensional Applications and Games.

In games, scene graphs are used to describe the structure of worlds and levels. All entities and objects in these worlds are part of a scene graph. Again here, the grouping and dependency mechanisms are used, so moving a single house should also move the interior and furniture with it. Scene graphs in games are in no way static. The game may define a relationship between an item and the character that is holding it. So when the character moves through the game world, so does the item that he holds. However, the item may be transferred to another character in the game, which removes the relation to the first character and adds a relation to the new owner of the item.

Usually scene graphs are just mathematical trees, however, in the increasingly large worlds of games memory requirements are an important factor. Due to this many scene graphs use instancing to reduce the space requirements. For example in a car model realized by a scene graph (see Figure 3.16), each wheel has it's own scene graph node, however, the graphical representation – the mesh, the textures and shaders – are only stored once and are referenced by all these individual nodes. So the mathematical tree becomes a directed acyclic graph.

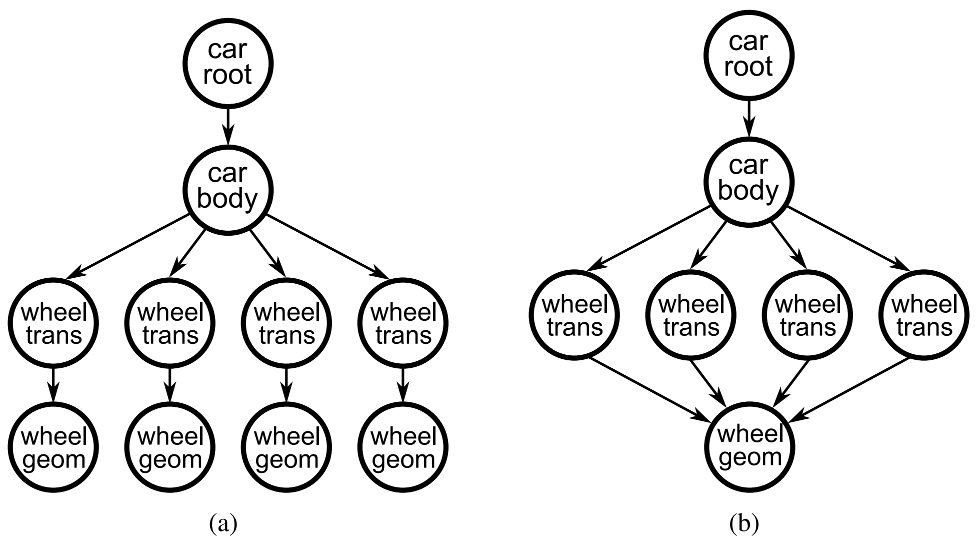


Fig. 3.16 Two scene graph representations of a car. Each wheel has its own transformation node for defining the position and orientation of them within the car mode. While the scene graph in (a) is a mathematical tree, it has a major drawback: the wheel geometry is stored for each wheel separately. The scene graph in (b) is a directed acyclic graph, where all wheel transformation nodes refer to the same geometry of the wheel stored in one node.

Alternatives To Scene Graphs. Especially in computer games, scene graphs are a tool that is often only used for the initial distribution of objects in a scene editor. While the game simulation is running, the scene graph is not maintained any more. Instead the parent-child relationships are defined implicitly though a physics engine and collision detection. This way relations between objects do not need to be updated in a graph, e.g. if an object falls from a table, it has no longer a relationship to the table because it is not touching it any more.

3.6.2 Scene Graph Libraries

Scene graph packages are for example OpenSG [RVN13] or Open Inventor [VSG13], which both realize the before mentioned instancing of objects in another way. This part will mostly focus on the scene graph concept realized by OpenSG 2, for which Hecher [Hec12] designed an operator set for the scripting language GML (see Section 4.1.2).

In OpenSG a different approach is taken in comparison to other scene graph systems. The functionality of a node is separated into two parts: a node and a *node core*. Nodes are only used to define the hierarchy and topology of the graph. So nodes store all children – if they have any – and have at most one parent. In the core, however, the important information, such as geometry or transformations, is stored. So the core actually describes the type of the node. Every node needs to have exactly core, but one specific core can be shared over many different nodes. This core sharing is for example used for the instancing of objects. In the example of the car (see Figure 3.17), the wheels are represented by different individual nodes, which all have the same core that is storing the graphical representation of the wheel object. This node and core concept does allow an easy realization of an operator interface for OpenSG scene graphs, which is discussed in context of procedural modeling with GML in Section 4.1.2.

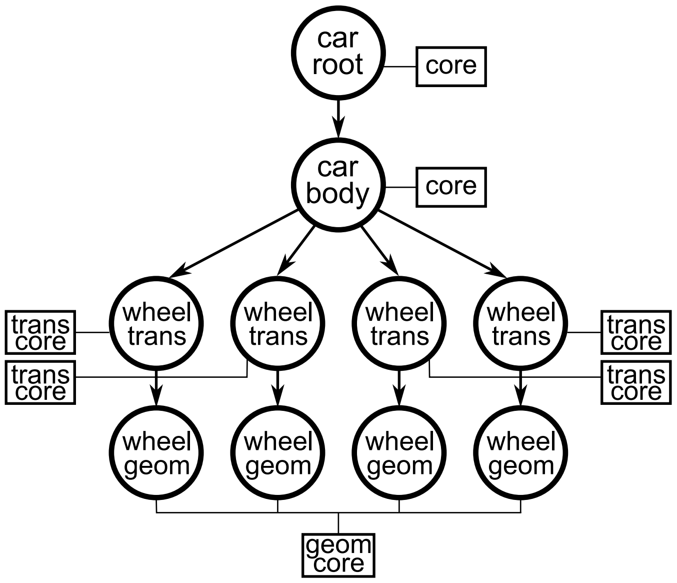


Fig. 3.17 Scene graph representation of a car in OpenSG. In OpenSG a node has additionally a core, which defines its type, attached to it. As in Figure 3.16(a), the scene graph has individual nodes for the wheel transformation and geometry. The transformations are all different, thus each node has its own individual core. However, the geometry nodes all share the same core, so that the graphical representation is only stored once, like in the example from Figure 3.16(b).

Managing Data in OpenSG. OpenSG is designed to handle multi-threaded data in an easy way. Copies of data are created on demand if that data is modified by multiple threads. This means, when more threads work on the same data, this data is simply shared; all threads reference the same data. However, at the time a thread modifies this data, a new independent copy of that data is created for this specific thread. So each thread carries its private copy (called *aspect*) of data to work on. At some point it is necessary to synchronize these different copies, to transfer all changed parts from one aspect to another. To do that, the system needs to keep track of all that has changed through the concept of *fields* and *field container*.

In OpenSG almost all data types are derived from field containers. These field container store data in the so-called fields. There are two distinct types of fields, *single fields* and *multi fields*. A single field, as the name suggests, stores a single value, whereas multi fields are comparable to dynamically resizing arrays. A node, for example, is a field container with a multi field for storing all children and a single field for storing the parent. The field container is the basic unit used for multi-thread safety. To keep track of all changes happening within a field container, each field container defines a constant for all fields it contains. These constants are used to create masks (by bitwise or), which then are provided to special beginEdit and endEdit functions, that keep track which fields have changed. By using this information data can be synchronized between multiple threads. For detailed information how these concepts have been translated to GML, please refer to the Master's thesis of Hecher [Hec12].

3.7 Constructive Solid Geometry (CSG)

Constructive solid geometry is another structuring method that brings shapes into a context. By combining several solids with Boolean operations complex objects can be achieved. A procedural description of the involved shapes and how they interact with each other increases the expressiveness even further. An in-depth introduction to the topic can be found in the book of Mäntylä [Män88].

The basic operations for CSG are Boolean operations on sets: union ($+$), difference ($-$) and intersection ($\cap$), which are illustrated with two simple shapes in Figure 3.18. Of course, these operations do not operate on the surface of shapes alone; they also need to take the interior of the shapes into account. Therefore, to ensure a coherent result, it is important that all shapes, which are combined with CSG, have a closed surface.

CSG methods can be applied in two or three dimensions and are often used by three-dimensional computer graphics and computer-aided design applications within the context of procedural modeling. CSG methods, together with free-form deformations (see Section 3.8), have been utilized in context of this thesis to describe curved and cartoonish buildings through split grammars (see Section 4.3 and 6.3.3). Furthermore, wedding rings have been procedurally described through the use of CSG operations (see Section 6.1).

The objects combined in through CSG are called *primitives*. Primitives can either be shapes that can be described by relatively simple parametric mathematical formulas like cuboids, cylinders, spheres, etc. or can also be complex shapes like arbitrary meshes that may have been created procedurally. In general, all primitives can either be procedural or not, but they need to be closed. By combining these primitives, objects with high complexity can be described by using only simple shapes to start with. Objects realized through CSG can be represented by a binary tree (see Figure 3.19). Leave nodes of this tree represent primitives and intermediate nodes represent the Boolean operations that combine the corresponding two sub-trees. Affine transformations can also be performed in each tree node.

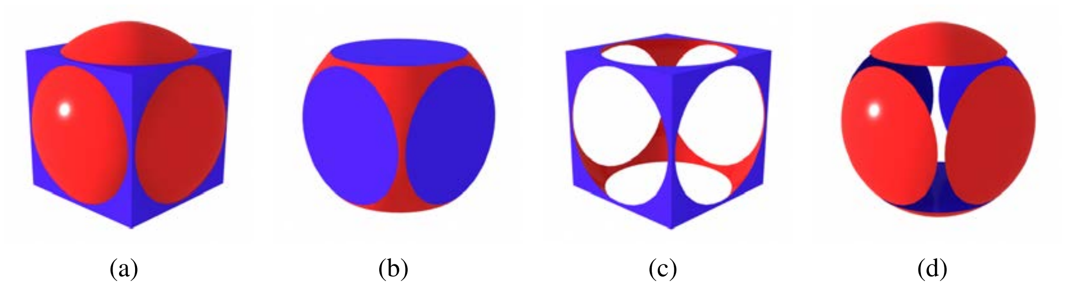


Fig. 3.18 Constructive solid geometry (CSG) operations between a blue cube A and a red sphere B: the union $A+B$ (a), the intersection $A\cap B$ (b), and the differences $A-B$ (c) and $B-A$ (d). Parts that belong to the cube are always colored blue and parts that belong to the sphere are colored red respectively.

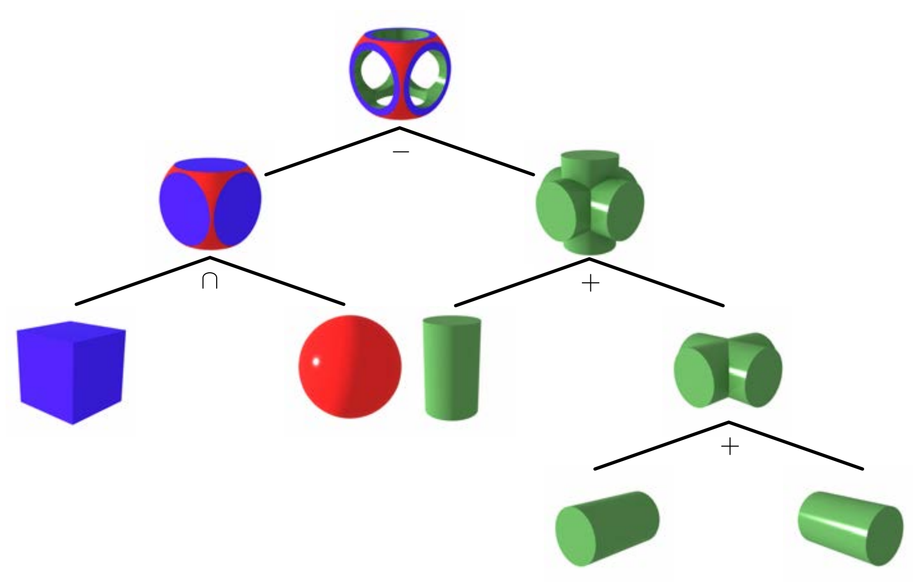


Fig. 3.19 A solid described by CSG operations can also be represented by a binary tree. Primitives are stores in leave nodes and the intermediate node represent the Boolean operations that combine the corresponding two sub-trees. The final object is illustrated at the root of the binary tree. The colored parts belong to the corresponding primitive: red for the sphere S, blue for the cube C and green for the cylinders C_z, C_x, C_y. The corresponding CSG formula is $(C\cap S)-(C_z+C_x+C_y)$.

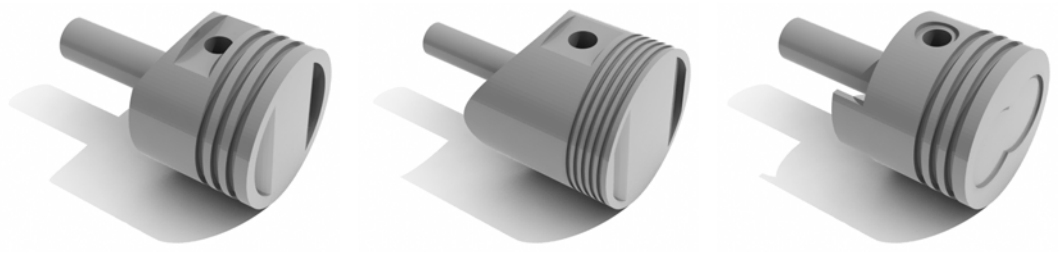


Fig. 3.20 Pistons of engines realized by combining procedural parts through CSG operations. Varying parameters and changing the positions of elements allows the generation of different results from one common description. (model courtesy of Johannes Edelsbrunner)

Properties of CSG. One of the advantages of CSG is that it is easy to determine whether the object is water-tight or not. If all underlying primitives are water-tight, then the combined shape has to be water-tight too. Such a check is of course important for manufacturing or engineering tasks. For arbitrary meshes, on the other hand, topological data or additional checks are required to assure that a given object is water-tight.

Another convenient property of CSG representations is that it is relatively easy to determine whether a point lies inside or outside a given object. The point is classified against all primitives that belong to the shape. By evaluating the corresponding Boolean expression the location of the point can be determined. The same way ray intersections can be done. The ray is intersected with all primitives and entry and exit point are calculated for these objects. This leads to one-dimensional intervals for each primitive along the ray, for which the Boolean expression can be evaluated easily. These properties are desirable qualities for applications like collision detection or ray tracing.

Computing the Results of CSG. To achieve CSG there are two different approaches: *Object space* approaches (see Section 3.7.1) and *image space* approaches (see Section 3.7.2). Object space approaches are for example analytic intersections of objects. Object space methods always yield a geometric primitive as result, which can be further used. Image space (or *screen space*) methods, on the other hand, do not create geometric primitives as a result. Image space methods are used to render the correct image on the screen. Common image space methods are ray-casting, scan-line or z-buffer algorithms. While calculations with object space methods have only to be done once, image space calculations have to be done frame by frame. However, the cost of object space method calculations is typically higher than frame by frame computations. When the relative position between objects used in Boolean operations change relatively often, it is wiser to use an image based approach. However, static models achieved through CSG can be precomputed using object space methods.

Procedural Modeling with CSG. In many domains CSG operations have significant advantages over traditional modeling methods. One such domain is computer-aided design. Objects manufactured through processes like milling or grinding are ideally described by Boolean operations. Here material is removed from initial created pieces, which are often cylinders or cuboids. A piston of an engine is an example where from initial pieces (cylinders) material is removed to form the final shape. Figure 3.20 illustrates the advantages of a procedural description of the separate parts. By varying parameters and changing spatial arrangements different results can be achieved from the same description.

Parameters are often changed within procedural modeling tasks. So if a procedural model consists of CSG operations, quick previews to see the changes in the model are desirable. For such tasks image space methods are more favorable than objects space methods because image space methods take less time to produce a visual result. Once the final model has been generated, object space methods can be used to produce a three-dimensional mesh that can be processed further.

An Operator Interface for CSG. An operator interface for CSG is quite trivial. An interface needs to support the three basic Boolean operations on sets: union, intersection, and difference. By combining simple shapes through application of these operators, shapes of high complexity can be described procedurally. These operators have been realized by Krispel [Kriar] for volumetric primitives, namely convex polyhedra (see Section 3.5), within the GML (see Section 4.1.2). Furthermore, I implemented operators for image space CSG methods in the GML, which have been used to visualize procedural deformed cartoonish buildings (see Section 4.3 and 6.3.3) and wedding rings (see Section 6.1) interactively.

3.7.1 Object Space CSG Methods

To calculate the a mesh of a procedural model involving CSG operations, object space CSG methods are necessary. However, while Boolean operations on objects, which are described by mathematical formulas, can be calculated accurately, Boolean operations on meshes suffer from accuracy issues due to the floating-point representation of the vertices. Accuracy errors are introduced at the moment a calculated vertex is expressed through floating-point values. However, predicates, such as whether a point lies above or below a certain plane, can be calculated exactly with appropriate data structures and computational effort. Bernstein and Fussel [BF09] implemented a fast, reliable method to compute Boolean operations on three-dimensional polyhedra using BSP-trees. Nevertheless, Boolean operations of two meshes always lead to a high computational effort, so a trade-off between speed and accuracy is necessary.

To obtain results as accurate as possible it is necessary to choose a fitting representation for the results obtained from the Boolean operations. A common representation for the mesh primitives used for Boolean operations are cellular structures, like binary space partition (BSP) trees, or convex polyhedra represented by half spaces (see Section 3.5).

The smallest family of solids that are closed under Boolean operations are *Nef polyhedra* [Nef78, BN88]. These are obtained from Boolean operations of half spaces. In contrast, convex polyhedra are obtained just by the intersection of a set of half spaces and are therefore a subset of Nef polyhedra. Nef polyhedra are very general because they are also able to express non-manifold solids as well as unbounded solids. This generality, however, is needed to describe Boolean operations between arbitrary meshes because non-manifold situations can occur for intersecting meshes. An implementation of Nef polyhedra by Granados et al. [GHH*03] is available within the *Computational Geometry Algorithms Library (CGAL)* [Cga13].

Further cellular structures that are able to express Boolean operations between arbitrary meshes with different strengths and weaknesses are presented by Rossignac and O'Connor [RO89] and Gursoz et al. [GCP90]. A detailed overview on other related methods is also presented in the paper of Granados et al. [GHH*03].

3.7.2 Image Space CSG Methods

Image space methods are ideal for a quick visualization of parameter changes within procedural models that involve CSG operations. All image space methods are view dependent. In contrast to view independent object space methods, the CSG operations need to be evaluated for each frame anew. These per-frame calculations are generally done much quicker than view independent object methods. This section reviews two different techniques for image space CSG methods, namely *ray casting* and *depth buffer methods*.

Ray Casting for CSG

Solving the logic of the Boolean operations can be simplified by only considering a ray in space at a time. For a ray, the closest visible surface that satisfies the CSG logic needs to be determined. This amounts to evaluating the Boolean operations of one dimensional intervals, which are spanned by the entry and exit points of the ray for the CSG primitives. The advantage of ray casting is that ray-object intersection can be calculated for a multitude of objects, ranging from mathematical descriptions to arbitrary boundary representations. Rays are generated for each pixel of the image to render. These rays can then be processed in parallel by the graphics hardware.

Depth Buffer Methods for CSG

Primitives are clipped by a multi-pass algorithm using the depth and stencil buffer. The main technique is based upon the algorithm introduced by Goldfeather et al. [GMTF89]. Their approach to render arbitrary CSG trees is based on three concepts: *surface parity*, *tree normalization*, and *surface clipping* using the depth buffer. An operator set for this depth buffer method has been developed by me for the GML. These operators have been used to describe deformed cartoonish buildings (see Section 4.3 and 6.3.3) and wedding rings (see Section 6.1) procedurally.

Determining the Surface Parity per Pixel. The *surface parity* for a pixel indicates whether a given primitive p is inside or outside of another given volume v. The parity value is achieved by counting the number of surfaces of p in front of the first surface of the volume v. The surface parity is calculated for each pixel using the depth buffer and is stored within the stencil buffer. Usually the volume v is first rendered to the depth buffer. Afterwards the primitive p is rendered without updating the depth buffer; instead for each pixel it is counted how many times the depth test would succeed. For regions of even parity the primitive p is outside of the volume v, which means that a ray cast through the volume of p enters and leaves p the same amount of times before hitting v. For regions of odd parity the intersection of p and v is not empty; the ray cast through p would hit v before leaving p.

There are several assumptions on p and v. First of all, all boundaries need to be closed. Missing triangles, due to an incomplete representation or clipping, yield wrong results. Additionally all participating surfaces should not be self-intersecting.

CSG Tree Normalization. Not every CSG tree is directly usable for image space CSG rendering. Most trees have to be converted into a suitable format first. A Boolean union operation that is combining primitives is called a *sum* and Boolean difference and intersection operations are called a *product*. A CSG tree that is in a "sum of products" form is said to be *normalized*. Normalized trees have the following characteristics:

- Union nodes (if there are any) appear only at the top of the tree.
- No node representing a Boolean operation is right of an intersection or difference node.
- No union node is a child of an intersection or difference node.

Tree normalization converts any CSG tree into a manageable format for depth buffer-based algorithms. The advantage of this normalization is that each product can be rendered by comparing the current result to one additional primitive instead to a whole sub-tree. Furthermore all union nodes can be neglected because the depth buffer automatically combines the corresponding products in the right way using the standard “depth less than” depth test. The normalized tree of the CSG tree in Figure 3.19 is shown in Figure 3.21(a).

Detecting the Pixels that Satisfy the CSG Logic. The process of rasterizing primitives into the depth buffer and detecting the pixels that satisfy the CSG logic of the products is referred to as *surface clipping*. After the tree normalization all intersection and difference operations are done sequentially, and at each step one new primitive is tested against all the ones that form the current state. Dependent on the operations, pixels of certain parity are retained. In case of intersection operations, depth buffer regions of odd parity, where the depth buffer values are volumetrically inside the primitive, are kept and all other regions are neglected. The inverse is happening for difference operations. Regions of even parity are retained and the remaining regions are discarded.

The orientation of the primitive surfaces is important when combining all clipped surfaces to a result. For image space methods only visible surface parts are considered and they form the result within depth and color buffer. The front or back faces of primitives are selected to be visible dependent on the operations the primitives take part in. For subtracted primitives the back faces are considered visible and for the other primitives the front faces are retained.

Figure 3.21(b) shows the clipped surfaces for all primitives (for the tree in Figure 3.21(a)), which are assembled within the depth and color buffer. The visible parts (front faces) are colored and invisible parts (back faces) are shown in black. Note that the orientation of the faces of the subtracted cylinders has been reversed.

Handling CSG with Non-Convex Shapes. The algorithm proposed by Goldfeather et al. [GMTF89] further introduces a way to deal with non-convex shapes. The problem of non-convex shapes is that overlapping parts may be cut away and no information about the geometry behind is available in the depth buffer. A shape is called *k-convex* if a ray can enter and exit the shape at most *k* times. A 1-convex shape is convex in the usual sense; a torus, for example, is a 2-convex shape. To process a *k*-convex shape, it has to be divided into up to *k* front facing and *k* back facing surfaces (dependent of the view direction less than *k* pairs might be necessary). These surfaces need to be processed individually. In each pass a portion of the shape’s surface is retained and assembled into the final color and depth buffers. This process is also called *depth peeling*.

Optimizations of CSG Algorithms. Goldfeather et al. also discuss advantages of pruning the CSG tree while normalization. Normalization of CSG trees may increase the size and amount of leaf nodes. In most cases whole sub-trees do not contribute to the final image because primitives within them do not intersect. To minimize the tree growth during normalization they use bounding boxes to determine whether primitives intersect or not. In large scenes this yields a big performance boost.

One inherent problem of this approach is that the contents of the depth buffer need to be saved and restored. This is not optimized in many hardware rendering environments. Steward and Leach [SLJ98] introduced an optimization by taking advantage of depth complexity. In general, each intersection and difference operation needs one rendering pass, which includes saving and restoring depth buffer content. However, often primitives do not overlap and can thus be clipped and rendered at the same time. By using this information, the number of passes

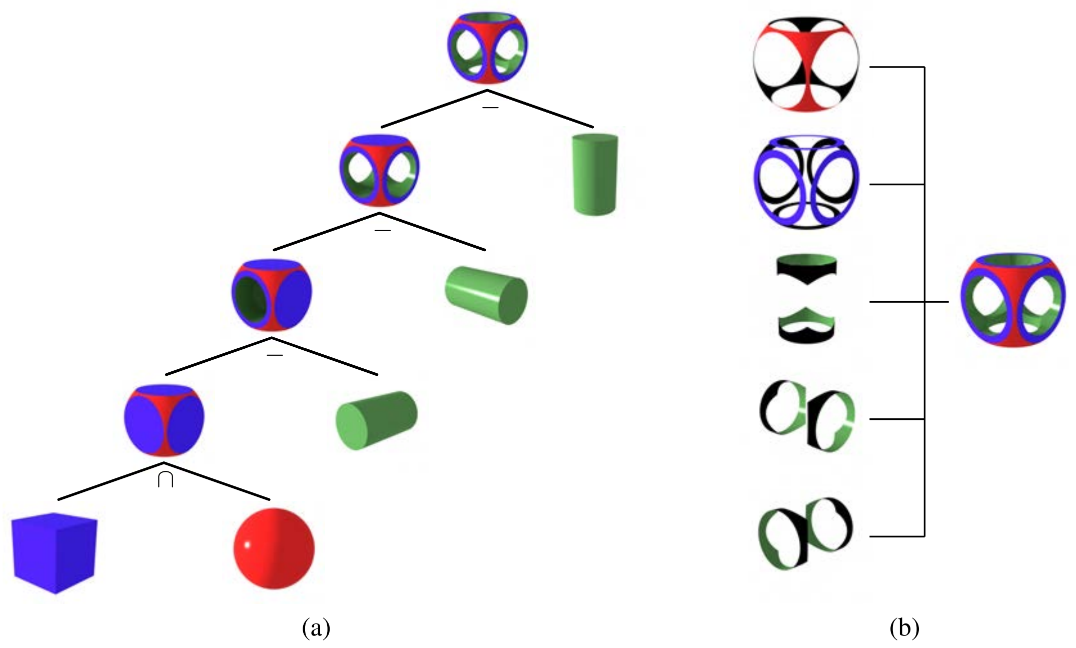


Fig. 3.21 Illustrations for rendering CSG in image space. The union of cylinders that is present in the CSG tree in Figure 3.19 has been split to normalize the tree (a). In normalized trees operations always add one additional primitive to the current state. This eases the rendering because no sub-trees have to be compared. The CSG formula changed to $C \cap S - C_x - C_y - C_z$, which is equivalent to the one in Figure 3.19. Based on surface parity and operations, the surfaces are clipped (b) and assembled within depth and color buffer. Not visible surfaces (back faces) are rendered in black. The orientation of the cylinder faces has been inverted because they take part in a difference operation, so the back faces become visible.

needed for rendering can be reduced. Concerning a single pixel, the term *depth complexity* refers to the number of primitives that cover this pixel. Based on the maximum depth complexity for a given view, they extract layers of primitives that can be processed at the same time. For n primitives the algorithmic complexity of the algorithm proposed by Goldfeather et al. is $O(n^2)$; the performance of the algorithm of Steward and Leach is $O(kn)$ for the maximum depth complexity k. There has been, however, no improvement for the worst case scenario ($n = k$).

3.8 Free-Form Deformations

Structuring methods do not always need to combine more shapes; they can also alter the appearance of a single shape. A way to change the appearance of a shape is to apply a deformation to it. Applying deformations to a mesh is an easy way of creating curved surfaces, provided that the surface has been tessellated sufficiently beforehand. Since deformations are simple parametric functions that can be applied to arbitrary meshes, they can be utilized within procedural modeling. In contrast to subdivision surfaces (see Section 3.4), deformed meshes are easier to handle, but require more computational effort, due to the higher tessellation required. In context of this thesis, free-form deformations have been utilized to describe curved architecture (see Section 6.2.3 and 6.3.2) and, together with constructive solid geometry methods (see

Section 3.7), cartoonish buildings (see Section 6.3.3) procedurally. This extension to the split grammar formalism is detailed in Section 4.3.

The Free-Form Deformation Algorithm. Based on an earlier technique by Barr [Bar84], the *free-form deformation*, introduced by Sederberg and Parry [SP86], is an approach where the space around a given model is deformed. The main idea is to envelop the mesh, which is subject to be deformed, with a geometrically easy to describe volume. Within this volume each point of the surface to deform can be parametrically defined explicitly. After the deformation of the enveloping volume, the new position of each point of the mesh to be deformed can be calculated by using the parameters describing the deformation and the initial parametrization of the vertices of the mesh. Consequently, the mesh gets deformed indirectly by deforming the space that surrounds it (see Figure 3.22).

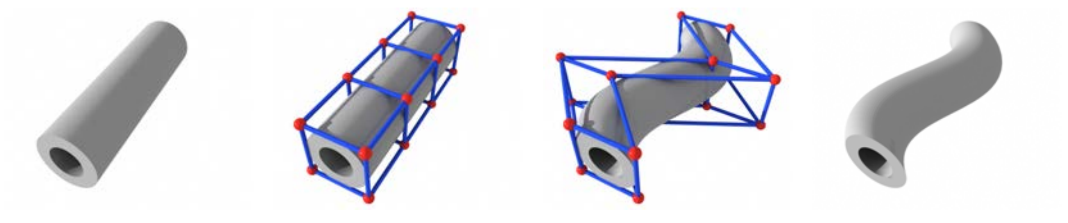


Fig. 3.22 The principle of a free-form deformation illustrated by a sequence of images. The axis-aligned bounding box of a straight tube (a) serves for the generation of a grid that envelops the tube (b). The control points (red) of this grid can be arbitrarily displaced (c) which leads to a deformation of space, which affects the shape of the tube (d).

In the algorithm of Sederberg and Parry this volume is described by a three-dimensional grid forming a cuboid, which is often the bounding box of the object to deform. This grid enables the use of a local Cartesian coordinate system to describe each point of the enveloped surface. The origin of this system usually lies within one corner point and the three outgoing edges of the cuboid are used as the axes. The amount of grid cells in each direction of the cuboid's axes define the degrees of freedom of the deformation. The algorithm of Sederberg and Parry was furthermore extended by Coquillart [Coq90] to support different hull objects such as cylinders and prisms.

Let us assume a three-dimensional coordinate system, like the one in the algorithm of Sederberg and Parry, with origin O, with the three normalized basis vectors $\vec{x}$, $\vec{y}$, and $\vec{z}$, as depicted in Figure 3.23. The *deformation grid* is composed of $l \times m \times n$ cells and has dimensions $X \times Y \times Z$. Each point P inside the grid can be described uniquely by:

$$P(s,t,u) = O + s \cdot \vec{x} \cdot X + t \cdot \vec{y} \cdot Y + u \cdot \vec{z} \cdot Z, \text{ for } 0 \leq s,t,u \leq 1. \quad (3.7)$$

For points outside of the grid at least one of the parameters s, t, or u has to be either less than zero or greater than one respectively. The parameters s, t, and u can be calculated for each point P by solving three linear equations, which amounts to normalizing each coordinate of the distance vector between P and the origin O with the respective extent of the deformation grid in each direction:

$$\begin{pmatrix} s \\ t \\ u \end{pmatrix} = \begin{pmatrix} \frac{(P-O)\cdot\vec{x}}{X} \\ \frac{(P-O)\cdot\vec{y}}{Y} \\ \frac{(P-O)\cdot\vec{z}}{Z} \end{pmatrix} \tag{3.8}$$

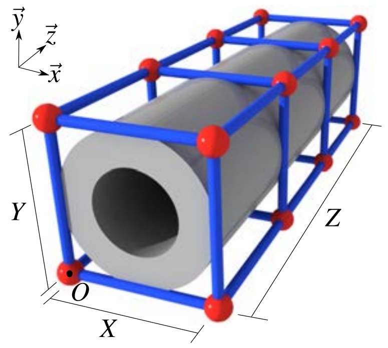


Fig. 3.23 A free-form deformation is a deformation of space achieved through displacement of control points that are initially placed on a regular grid around the object to deform. In this case the object to deform is a straight tube and the deformation grid is based on the axis aligned bounding box of the tube. The grid of dimensions $X \times Y \times Z$ has its origin in O and is composed of $1 \times 1 \times 3$ cells.

The deformation approach is based upon Bézier splines [Far97] and their generalization to the third dimension. The basis functions are the so-called *Bernstein polynomials*.

Definition 3.13 *The* ***Bernstein polynomials*** *of degree n are defined by*

$$B_{i,n}(t) = \binom{n}{i} \cdot t^i \cdot (1-t)^{n-i} \tag{3.9}$$

for $0 \leq i \leq n$ *and* $0 \leq t \leq 1$.
These polynomials have following properties:

- *The sum of all Bernstein polynomials evaluates to one:*

$$\sum_{i=0}^{n} B_{i,n}(t) = 1. \tag{3.10}$$

- *Each single polynomial has a value between zero and one:*

$$0 \leq B_{i,n}(t) \leq 1. \tag{3.11}$$

The deformation grid is described by *control points* $K_{i,j,k}$ that are used to describe the grid cells. In a one dimensional case, the position of a point on a Bézier curve is defined by a weighted sum of control points K_i with the weights being the corresponding Bernstein polynomials $B_{i,n}$:

$$P(s) = \sum_{i=0}^{n} B_{i,n}(s) \cdot K_i, \text{ for } 0 \leq s \leq 1. \tag{3.12}$$

This formulation can be easily extended to arbitrary many dimensions. For our cause three dimensions is sufficient. This leads to:

$$P(s,t,u) = \sum_{i=0}^{l}\sum_{j=0}^{m}\sum_{k=0}^{n} B_{i,l}(s) \cdot B_{j,m}(t) \cdot B_{k,n}(u) \cdot K_{i,j,k}, \text{ for } 0 \leq s,t,u \leq 1. \tag{3.13}$$

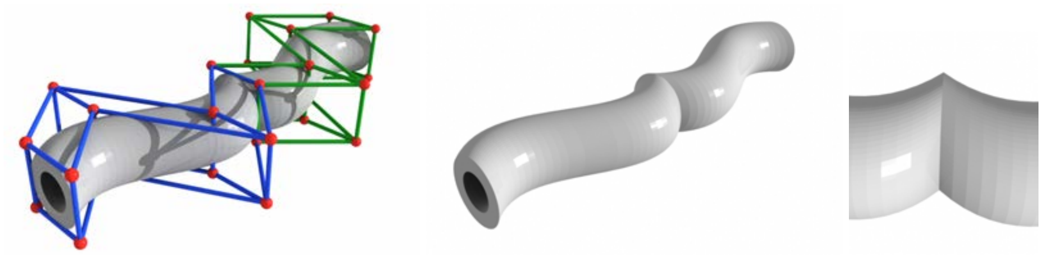
A point P on the mesh to deform is therefore expressed by a linear combination of basis functions and control points. If the control points $K_{i,j,k}$ are not altered, then the formula in Equation 3.13 is an identity transformation.

However for a free-from deformation to achieve suitable results the tessellation of the source mesh has to be quite high to realize a smooth surface after deformation. Free-from deformations do not introduce new points while deforming, but adaptive refinement methods can use used to tessellate the surface in specific regions for local deformations or overall to meet certain accuracy conditions.

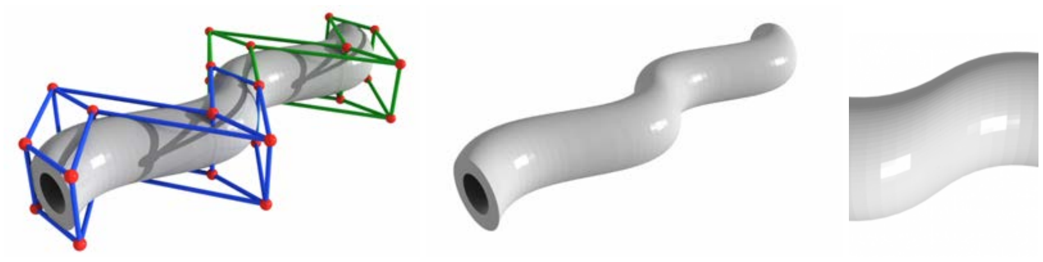
Properties of Bézier Splines. As free-form deformations are a generalization of Bézier splines in three dimensions they inherit all limitations and advantages from them. Important properties of Bézier splines are that the number of control points can be increased without changing the curve. This can be used to split one spline into two different curves. These features allow local control of deformations.

Depending on the layout of the control points, Bézier splines can join with any continuity. Two distinct Bézier splines, however, cannot in general be fused into a single curve, even if they join with high continuity. Let us take two splines f and g, with control points $f_1, \ldots, f_i$ and $g_1, \ldots, g_j$. For C^0 continuity the end point of the first curve f_i needs to coincide with the starting point of the second curve g_1. To reach C^1 continuity additionally the tangents (first derivative) of the curve at the touching point $f_i = g_1$ need to coincide. This amounts to the three points f_{i-1}, $f_i = g_1$, and g_2 being collinear and equidistant. Finally, to reach C^n continuity on the boundary between two curves, the first to n^{th} derivative needs to coincide at the touching point $f_i = g_1$. Reaching high continuities does impose much on the control point design, especially for free-form deformations because these constraints must be met for two touching three-dimensional grids (see Figure 3.24).

An Operator Interface for Free-Form Deformations. An operator interface to create free-form deformations of shapes procedurally basically needs to provide only two kinds of operators. First, an operator for encapsulating shapes with control point grids is needed. This operator has to define the origin, the dimensions, as well as the amount of cells in all three major axis directions of the control point grid. Second, an operator is needed that allows displacing of control points. With these two operators it is possible to apply deformations (that can even be nested) to shapes. An operator interface for free-form deformations that has been included in the scripting language GML is explained in Section 4.1.2. Free-form deformations have, furthermore, be introduced in the split grammar formalism (see Section 4.3) to describe curved architecture (see Section 6.2.3 and 6.3.2) and cartoonish buildings (see Section 6.3.3) procedurally.



(a)



(b)

Fig. 3.24 Two deformed shapes can only join with a continuity higher than C^0 through an appropriate control point grid design. The naïve approach (a), in which only the points of the grid coincide at the transition point, leads to artifacts in form of creases. Without loss of generality let us assume that we want to reach a transition with C^1 continuity in x direction. To achieve this, the control point grids need to be positioned in a way that the points, which are adjacent to the coinciding points in x direction, need to be collinear (b). This assures that the first derivatives of both deformations are the same at all the touching points, which ensures a smooth transition.

Synopsis

In this chapter I introduced a hierarchy of shape representations. At the lowest level are the low-level shape representations, which include – among others – point clouds, different kinds of meshes, subdivision surfaces, spline surfaces, or volumetric representations such as convex polyhedra or union of spheres. All these representation are used to describe the shape itself. On the next level are structuring methods, which either set shapes into context or alter their appearance. Out of this group scene graphs, constructive solid geometry, and free-form deformations have been discussed in this chapter. On the top of this hierarchy are procedural methods, which use representations from the lower levels to create shapes procedurally.

The goal of this chapter was to show how shape representations can be made amenable to the procedural approach. This is done by providing a set of operators, which describe the generation of shapes and structures of the corresponding representations. In this chapter I introduced potential operator sets for all discussed shape representations, which are utilized by the procedural techniques that are presented in the upcoming chapter.

Chapter 4
Procedural Technologies

Contents

Abstract. This chapter focuses on procedural technologies that have been used and developed in the context of this thesis. These technologies make use of the low-level shape representations and structuring mechanisms discussed in the previous chapter. This chapter features an introduction to the generative modeling language GML, as well as important modeling vocabularies of that language. Additionally, the split grammar formalisms, which have been an extension of the state of the art of procedural architecture, are explained in detail in this chapter. This includes split grammars based on a more general class of shapes, namely convex polyhedra, and split grammars that incorporate free-form deformations into the split grammar formalism in a way that rules can adapt to them. Furthermore, the GML Compositor, an interactive procedural modeling tool, is presented together with several applications in different modeling domains. This tool provides a procedural modeling context without the need of scripting and visualizing the underlying data flow graph.

4.1 The Generative Modeling Language – GML

This section introduces the Generative Modeling Language (GML), which was presented by Havemann in his Ph.D. thesis [Hav05] in 2005. It is inspired by Adobe PostScript (see Chapter 3 from the PostScript Language Reference [Ado99]), but is rather for describing three-dimensional shapes than two-dimensional images and has been designed for procedural modeling. Thus, the GML can also be efficiently used as file format for low-level shape descriptions. It, furthermore, comes with an integrated visualization engine. Due to these features, the fact that GML is developed at our institute, and the rich catalog of existing procedural modeling libraries (see Section 4.1.2), GML has been used for procedural modeling in this thesis.

Executables for interpreting the language and software demos are available at the homepage: `http://www.generative-modeling.org/`. Furthermore, a full list of available operators is available at the GML Wiki [Ins13].

4.1.1 Introduction to the Generative Modeling Language

The GML is a simple stack based scripting language. A GML program is not parsed but interpreted as a stream of tokens that are evaluated sequentially. A *token* is either a *literal*, which represent data that is pushed onto the stack, or is an *operator*, which is executed. The stack – more precise, the *operand stack* – is used to provide the inputs for every operation executed. These functions pop their inputs from the stack and push their results onto the stack after execution. These results then serve as inputs for the next operations. So the job of the GML interpreter is just to provide data that was output from one function to the next function executed. The GML offers two different kinds of operations, on the one hand, *atomic operators*, which are built-in and are implemented in C++, and *composed operators*, which consist of data and calls to atomic operators and other composed operators. Just like Adobe PostScript generates bitmaps with side effects of operations, GML uses these side effects to manipulate internal states and generate three-dimensional shapes.

GML Tokens and the Tokenizer. In GML almost any string can be tokenized. The tokenizer generally uses white spaces as delimiters. There are a variety of different atomic data types: integer values (10), floating point values (0.5), strings ("test"), two- and three-dimensional vectors ((1.0,1.0) and (1.0,1.0,1.0)), and *literal names* (/name) that are identified by their respective slash prefixes. There are also special marker literals [, { and }, which are used to produce different kind of arrays.

New tokenizer rules can be added on C++ level. Other built-in token types are dictionaries, arrays, operators, path names, and executable names. These token types will be explained later in detail. Tokens have a fixed internal size of 16 bytes; consequently, for big data, or data with variable size, only a reference to the actual data can be stored within the token.

Execution and Notation of GML Code. Instead of the familiar infix notation ($(3+4)\cdot 6$ and func(a,b,c) respectively), a postfix notation (3 4 add 6 mul and a b c func respectively) is used because this notation is a natural fit for a stack based language. Infix notations need brackets to make up for the precedence of the multiplication over the addition, which is used to resolve ambiguities within that notation. In postfix notations all operations are equally treated and therefore an expression can be resolved uniquely. This feature made postfix-based notations popular; first of all pocket calculators used this notations.

Take the program 3 4 add 6 mul as an example. When this code is tokenized, five tokens are generated, which are executed one after another. The first two tokens 3 and 4 are literals, so they are pushed to the stack. The next token add is an executable name, which is looked up in the dictionary stack. There the atomic add-operator is referenced under this name. The add-operator pops two tokens (3 and 4) from the stack and pushes the result ($3 + 4 = 7$) onto the stack when executed. Afterwards the literal 6 and the executable name mul are processed which leads to the result $7 \cdot 6 = 42$ that remains on the stack after execution of this program.

Dictionaries and the Dictionary Stack of GML. The GML features another kind of token, namely *dictionaries*. A dictionary is a list of key and value pairs, where the key – being the unique identifier – is represented by a literal name, and the value can be represented by any kind of token.

The GML also utilizes a *dictionary stack*, which is, as the name states, a stack of several dictionaries that is used for look-up purposes. An *executable name* is a name which serves as key within these dictionaries. At the time when an executable name is interpreted, the corresponding value of the topmost dictionary on the dictionary stack, which contains this name as key, is executed. Dictionaries can be pushed to this stack with the begin operator; the topmost dictionary is popped from the stack with the end operator. The dictionary stack is never empty; there is always a *global dictionary*, which is used for look-up of the atomic operators.

Dictionaries can also be used to realize object-oriented programming within GML. A dictionary can be interpreted like a class that is editable and extensible during runtime. Dictionary hierarchies can be traversed using path names that are represented by their dot prefix. The dot prefix simply pops a dictionary from the stack and pushes the one found under the name following the dot. This enables traversing dictionary hierarchies like body.head.face, as it is common in C++.

Array Types in GML. The [marker literal indicates the beginning of an array. In contrary, the closing bracket] is not a literal, but an operator that creates a literal array from all tokens that were pushed to the stack after the last [literal. Arrays are heterogeneous and, therefore, the carried literals are not bound to any type constraint. Consequently, a valid array is for example [1.0 /two "three" (4,5,6) [7]].

The { and } literals are used to create another kind of array, namely *executable arrays* respectively *functions*. The { marker sets the interpreter to *deferred mode* so that all tokens are treated as literals and pushes them to the stack. Operations are not executed, but their executable name is pushed to the stack. The matching } marker ends this mode and the resulting executable array is pushed to the stack, but not executed. Executable arrays can also be nested.

The difference between these two array types is that normal arrays only carry the tokens that are on the stack at the moment when the] operator is executed. Executable arrays on the other hand carry the information on how these tokens are generated. As comparison the code [1 1 add] yields [2] on the stack, whereas the code { 1 1 add } yields the same executable array { 1 1 add } on the stack, which when executed pushes 2 to the stack. Executable arrays can therefore be used as function which take inputs from the stack, for example { dup mul } implements a simple power of two function.

For arrays a special aload operator is provided, which pushes each element of the array separately onto the stack. The flatten operator, on the other hand, is used to resolve nested arrays. All elements in a hierarchy of arrays are transferred to a single array in respect to their order.

The Register Extension of GML. A serious disadvantage of Adobe PostScript as programming language is the fact that it is hard to keep track of the stack. In Adobe PostScript two ways of storing variables are supported. One way is on the stack, where keeping track can become very tedious, and in dictionaries, which is relatively slow. The GML offers a third alternative to overcome these shortcomings, namely *registers*. They are, compared to dictionaries, very fast and efficient. The main purpose of registers is to provide functions with fast local variables. Thus, registers are only valid in the scope, respectively function, within they are defined.

Registers are enabled with the usereg operator at the beginning of a function. Values are stored within a name register with an ! as prefix, and are retrieved with either ; or : as prefix. The difference between ; and : is that the prefix ; just pushes the value on the stack, while the prefix : also executes the token stored within the register. Therefore the program { usereg !x :x } is equivalent to { usereg !x ;x exec }. The distinction between ; and : is only important when executable arrays are stored within registers, otherwise there is no difference.

Program Flow Control in GML. Conditional branches and loops are a necessity for any programming language. In the GML it is no different, but no special syntax or keywords are required like in other programming languages like C++. In the GML special *flow control operators* are used instead.

For example the branching operator (if) works just like any other ordinary atomic operator and operates on token of the stack. The signature is *flag procedure* if. It works like anybody would expect: the operator takes two tokens from the stack, and executes the first one (the *procedure*) if and only if the second one (the *flag*) is a value different to zero. It is important that the token representing the *procedure* does not necessarily needs to be an executable array; it can be any kind of token.

The complete list of available control flow operators is presented below.

flag procedure if $\longrightarrow$
The token *procedure* is executed if and only if *flag* $\neq 0$.

flag procedure1 procedure2 ifelse $\longrightarrow$
The token *procedure1* is executed if and only if *flag* $\neq 0$, *procedure2* is executed otherwise.

procedure loop $\longrightarrow$
The token *procedure* is continuously executed until the exit operator is called from within *procedure*.

amount procedure repeat $\longrightarrow$
The token *procedure* is executed *amount* times.

from by to procedure for $\longrightarrow$
The values $\mathit{from} + i \cdot \mathit{by}$, for $i \geq 0$ are pushed to the stack as long these values are all $< \mathit{to}$ (for $\mathit{by} > 0$), or $> \mathit{to}$ (for $\mathit{by} < 0$). The token *procedure* is executed each time. These values are on top of the stack upon execution and are used to imitate the control variable of usual for loops.

array procedure forall $\longrightarrow$
Each element from *array* is pushed to the stack and the token *procedure* is executed for each separately.

array1 procedure map $\longrightarrow$ *array2*
Each element from *array1* is pushed to the stack, afterwards processed by executing the token *procedure*, and then the result is popped from the stack. The result *array2* contains the processed values and is pushed to the stack afterwards. It is important that the token *procedure* must push a result to the stack for each processed value, so that the token *array2* can be constructed.

It is important to notice that only the map operator has an official return value, but any *procedure* token used with a control flow operator may leave values on the stack.

To create the *flag* values common comparison operators are provided. For the sake of completeness, these are: "equal" (eq), "not equal" (ne), "greater than" (gt), "greater or equal" (ge), "less than" (lt), and "less or equal" (le). If the expression evaluates to true an 1 is pushed to the stack and 0 otherwise.

4.1.2 Procedural Modeling with the GML

The GML can be used to model all kinds of geometry procedurally. Internally the GML is composed of so-called *resources*, modules that encapsulate certain domains and provide operators for them. This section focuses on important ways for modeling within the GML. The GML provides, among many others, resources for *combined B-reps*, which combine classical polygonal meshes and free-from surfaces, *convex polyhedra*, which are used for example for split grammars, and *scene graphs*, which are used to describe hierarchical scenes. In addition, an extension to utilize convex polyhedra in combination with *free-form deformations* will also be presented. The following parts will focus on how to create procedural models using these different modeling vocabularies in the GML by explaining the most important operators. An explanation of all operators is available within the GML Wiki [Ins13].

Procedural Modeling with Combined B-Reps in GML

Combined B-reps use triangulations to represent polygonal faces and use subdivision surfaces for the representation of curved parts and have been implemented in GML by Havemann. So a shape can consist of polygonal parts and free-form parts. At the core, the low-level Euler operators (see Section 3.3.2) are utilized. As modeling with these operators is very tedious and unintuitive, a set of high-level modeling operators that speed-up the modeling process are provided additionally within the GML. These high-level operators combine the low-level Euler operators into a set of intuitive modeling operations. Due to the high number of modeling operators an explanation of each is infeasible at this point.

The modeling toolkit focuses on combining subdivision surfaces with polygonal meshes. Polygonal meshes constructed with the available tools can be traversed using a halfedge data structure (see Section 3.3). Halfedges on the stack are visualized by arrows (as in Figure 4.1). The halfedge, which is on top of the stack, is shown as a red arrow. The mesh traversal operations like faceCCW or vertexCCW are all available in GML and take a halfedge as input and leave a new halfedge on the stack to highlight the current position on the mesh.

To achieve subdivision surfaces (see Section 3.4) a polygonal mesh is used as control mesh. Creases are set per edge in this mesh. In Figure 4.1 green edges symbolize soft edges and red edges symbolize hard edges. If just hard edges are present in the mesh, the output mesh is identical to the control mesh.

Modeling Example with Combined B-Reps in GML. To illustrate the handling of the operators presented in this section, a small code example is discussed here. The following code example (which is taken from [Hav05]) is augmented by the images in Figure 4.1, which illustrate the modeling process step by step. For a better understanding of the code, the following paragraph explains the separate code lines.

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

1: First the default color is set to cyan, so all objects modeled after this are colored in cyan. The quad operator reads as first parameter a mode flag 2 which determines the amount and interpretation of additional parameters. In this case the further parameters are the midpoint (0,0,-2) and the extent (1,1,0) of the quad.

2: With poly2doubleface a double-sided face is generated out of a polygon (array of points) and a mode flag, which defines the sharpness of the edges. A halfedge of the generated mesh is pushed to the stack (red arrow).

3: The extrude operator extrudes this face along its normal. As input it takes a halfedge and an offset vector, which contains horizontal and vertical offset as well as an sharpness flag. This operator pushes the halfedge of the extruded face to the stack.

4: The face vertices are moved by projecting it in z-direction (0,0,1) onto a plane (defined by normal and offset) by the project_ringplane operator.

5, 6: A second double-sided face in color yellow is generated on another position in the same way as in the lines 1 and 2.

7: The quad faces that are represented by the halfedges on the stack are connected with smooth edges by the bridgerings operator, which takes the two halfedges and a sharpness flag as input.

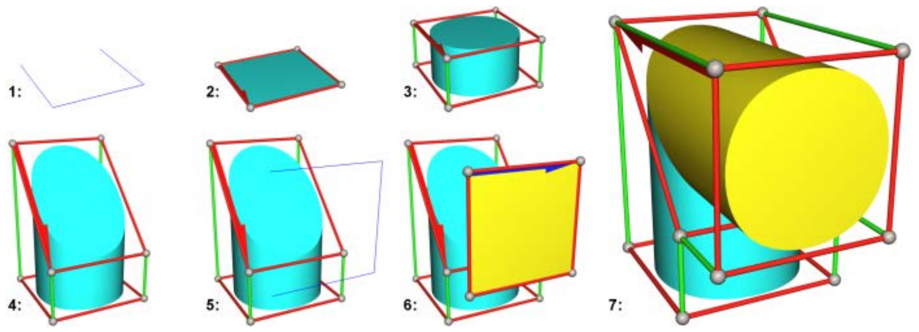


Fig. 4.1 Example for modeling with Combined B-Reps in the GML. The separate images illustrate the step by step execution of the code example provided in this section. (image source: Havemann [Hav05])

Complex Examples with Combined B-Reps Realized in GML. Aside from the toy examples, complex examples are presented in Figure 4.2. The cathedral in Cologne (Germany) together with corresponding gothic windows can be described efficiently with the GML. One of the important operators is the extrudestable operator, which implements a straight skeleton algorithm [AAAG95] and is able to extrude any face along of a profile with a different sharpness for each segment. This way, different profiles can be applied to the borders of the procedural Gothic window.

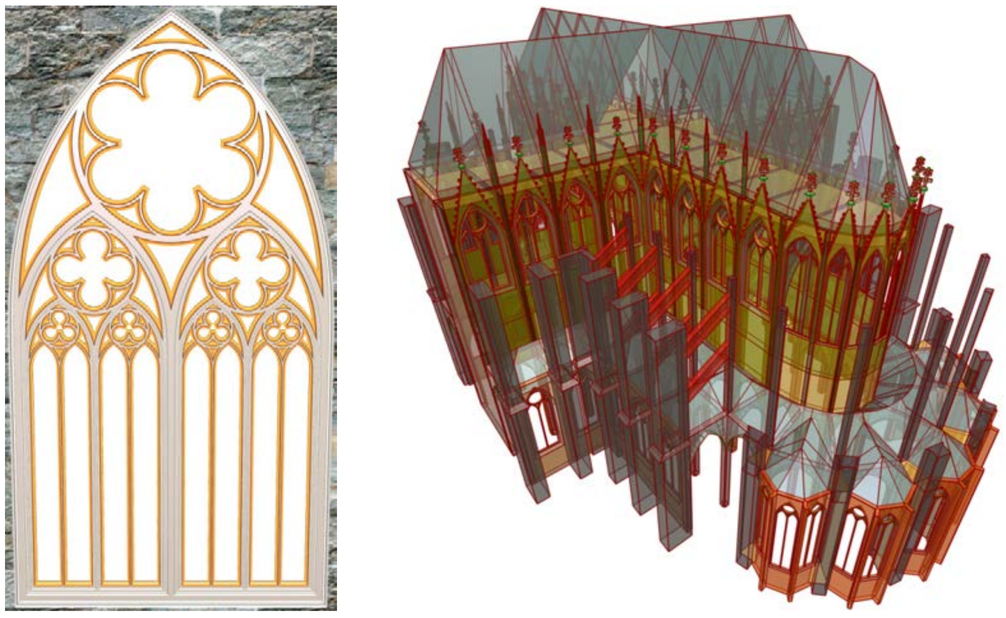

(a) (b)

Fig. 4.2 Two examples for modeling with combined B-Reps in the GML. The procedural Gothic window (a) is constructed using intersections of circle segments and styles are inserted recursively into the constructed sub-parts. Different styles of moldings are achieved with profile extrusions based on a straight skeleton algorithm. Windows like these were used in a procedural description of the cathedral in Cologne (Germany) (b). This cathedral is modeled using high-level GML operators. This amounts to the use of roughly 44.000 Euler operators. (image source: Havemann [HF04, Hav05])

Procedural Modeling with Convex Polyhedra in GML

Convex polyhedra as presented in Section 3.5 are used within the GML and were integrated by Krispel [Kriar]. This section will elaborate the most important operators necessary to create convex polyhedra in the GML.

All operations provided for modeling with convex polyhedra share the prefix cp- and also a second specialization prefix whether they operate on planes (pl-) or polyhedra (po-).

Basic Operators for Convex Polyhedra in GML. To avoid rounding errors that would occur with floating point values used for representing normals and points, the resolution of both is reduced. The operators for the conversion between the floating point representation and the reduced integer representation are cp-world2int and cp-int2world respectively.

Plane Operators for Convex Polyhedra in GML. A convex polyhedron is defined by a set of planes, of which each separately defines a half space. So, to provide a set of functions for generating convex polyhedra procedurally, operators for generating and operating on planes are necessary. There are two plane registering operators: cp-pl-register and cp-pl-regptnormal. The first creates a plane out of three vertices that are arranged in counter-clockwise order and the latter uses one explicit point with a normal to register a plane.

After a plane has been registered, it can be further used as reference to create new planes or be modified. Most important here is the cp-pl-flip operation, which takes a plane and pushes the flipped plane, which represents the inverse half space, onto the stack. The operators cp-pl-movept and cp-pl-movedist both calculate a new plane based on the provided one. The first returns a new plane with the same normal vector which goes through an additionally provided point; the latter creates a new plane with the same normal vector which is moved a given distance along the normal vector. Additionally, query operators are provided to retrieve information stored in planes such as cp-pl-getnormal to push the plane's normal vector onto the stack.

Polyhedron Operators for Convex Polyhedra in GML. With operators defined for handling planes, convex polyhedra can be created by providing an array of planes. The cp-po-register operator takes an array of planes and creates a convex polyhedron from it. For convenience there also exist operators like cp-po-registerbox, which create an axis-aligned box-shaped polyhedron based on two extremal points. Polyhedra can be further edited by adding planes (cp-po-addplane), which is equivalent to a trimming operation, or by removing planes (cp-po-removeplane), which enlarges the polyhedron. Duplicates of a polyhedron can be created with the operator cp-po-dup. By adding a plane and its flipped counterpart to two instances of the same polyhedron, a split can be achieved.

For polyhedra there is also a set of query operations, such as cp-po-getplanes, cp-po-getvertices, or cp-po-getfaces to return an array of planes, vertices, or polygons, respectively. More specialized query operations, for example, take a plane as input to retrieve the vertex with minimal signed distance to this plane (cp-po-getnearest).

Due to the plane-based representation Boolean operations of convex polyhedra can be handled robustly. The operations cp-csg-difference, cp-csg-union, and cp-csg-intersection each take two arrays A and B of polyhedra as input and return an array that is either the difference $A - B$, the union $A + B$, or the intersection $A \cap B$ of those input arrays. For an example see Figure 4.3.

Modeling Example with Convex Polyhedra in GML. The handling of convex polyhedra is illustrated by a simple code example that splits a polyhedron in two disjunct convex polyhedra. The example is explained step by step in the following paragraph.

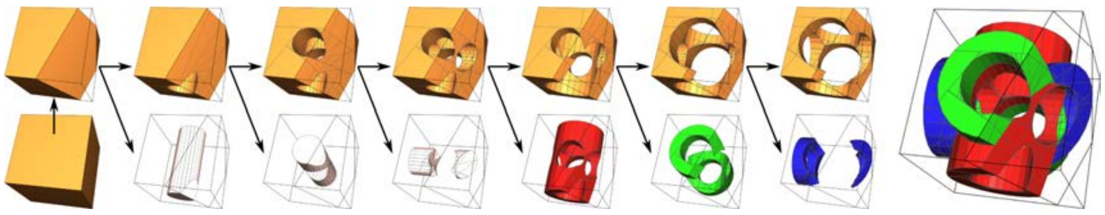


Fig. 4.3 Successive use of constructive solid geometry operations based on convex polyhedra in the GML. Starting from a cube trimmed by a plane, first smaller, then larger cylinders are intersected successively with the remainder of the cube. A union of three colored crossing pipes is created as a result. The representation is robust to not create any kind of artifacts.

```
1: (−2,0,0) (2,1,1) cp-po-registerbox !box1
   ;box1 cp-po-dup !box2

2: ;box1 /stdBlue (0,0,0) 1 1 1 cp-po-setproperties
   ;box2 /stdRed  (0,0,0) 1 1 1 cp-po-setproperties

3: (0.5,0,0) (1,1,0) cp-pl-regptnormal !plane

4: ;box1 ;plane             cp-po-addplane
   ;box2 ;plane  cp-pl-flip cp-po-addplane
```

1: An axis-aligned box-shaped convex polyhedron is generated between the points $(-2,0,0)$ and $(2,1,1)$ by the cp-po-registerbox operator. A copy of this polyhedron is generated by the cp-po-dup operator and is saved in another register.
2: With the operator cp-po-setproperties internal properties can be set for different convex polyhedra. These properties are in order: face material, outline color, an integer to save arbitrary references, and two Boolean flags that either enable or disable outlines and faces. Here one polyhedron is rendered red and the other blue; both have a black outline.
3: Based on a point and a normal vector a plane is registered.
4: The before created plane is added to both polyhedra, but once it is flipped before. This way, the original polyhedron is split into two (see Figure 4.4).

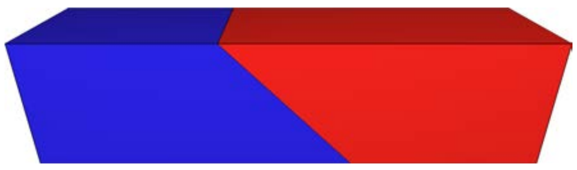


Fig. 4.4 The convex polyhedra created by splitting an initial polyhedron into two. This is the result produced by the code example presented in this section.

Procedural Modeling with Free-Form Deformations in GML

Free-form Deformations (as described in Section 3.8) have been integrated into the GML by me. This section covers the most important operators necessary to utilize free-form deformations in the GML.

At the core of the implementation lies a *deformer*, an object that carries the geometry to deform (which is usually composed of convex polyhedra), and a set of *modifier*, which define the visualization of the geometry. There are two different kinds of modifier: affine transformations and deformations, which are all applied – in order they were added to the deformer – to the geometry.

The basic operator for the use of deformations in the GML is the st-addDefomer operator, which takes an array of convex polyhedra as input. These polyhedra compose the geometry; further polyhedra can be added by the st-insertPolyhedron operator, which takes the deformer and the new polyhedron as input.

Modifiers are added to the deformer with the st-addModifier operator. The different modifiers for this operator are generated with st-registerDeformation and st-registerAffineTrans, respectively. Deformations are registered by a vector, which specifies the dimensions of the deformed space with origin in $(0,0,0)$, and three integers that specify the amount of control points in all three axis directions. Offsets can be applied to these control points through the st-deformAll operator, which takes a deformation token as well as an array of three-dimensional offsets for all individual control points as input.

Affine transformations, on the other hand, do not need any inputs for registration. Transformations can be set by the st-addTranslation, st-addRotation, and st-addScale operators. Additionally to the affine transformation token, translations are specified by a vector, rotations by a vector and an angle, and scales by a vector, respectively. Alternatively, the st-mulMatrix operator can be used to apply any transformation matrix. This operation takes – additionally to the modifier token – an array that carries 16 floating-point values. These values need to be specified in column-major order.

Another important operator is the st-deformPoint operator, which applies all modifiers applied to a deformer to a single point. This way, single points can be deformed for more delicate calculations.

Modeling Example with Free-Form Deformations in GML. To illustrate the usage of the aforementioned operators a small code example is presented. Again this example is discussed step by step in the following paragraph and intermediate results and the final result are shown in Figure 4.5.

```
1: [  (0,0,0)  (8,4,2)  cp-po-registerbox ]
   0 st-addDeformer !deformer

2: (8,4,2)  2 4 2 st-registerDeformation !deformation1
   (8,8,2)  2 4 2 st-registerDeformation !deformation2

3: [
     (0,0,0)  (0,0,0)
     (0,0,1)  (0,0,1)
     (0,0,1)  (0,0,1)
     (0,0,0)  (0,0,0)

     (0,0,0)  (0,0,0)
     (0,0,1)  (0,0,1)
     (0,0,1)  (0,0,1)
     (0,0,0)  (0,0,0)
   ]  !d
```

```
4: ;deformation1 ;d  st-deformAll
   ;deformation2 ;d  st-deformAll

5: st-registerAffineTrans !trans
   ;trans  (0,4,0)  st-addTranslation

6: ;deformer ;deformation1  st-addModifier
   ;deformer ;trans   st-addModifier
   ;deformer ;deformation2  st-addModifier
```

1: First the deformer is created by the st-addDeformer operator. It takes an array of convex polyhedra and an additional flag, which would invert the geometry, as input. The geometry in this case is composed of a single box-shaped polyhedron. (see Figure 4.5(a))
2: Two deformations of different dimensions $(8,4,2)$ and $(8,8,2)$ are created here. Both, however, share the same amount of control points in all three axis directions, which are two in x and z direction and four in y direction.
3: The offsets for all the individual control points are defined in an array.
4: The same offsets are applied to the two deformations of different sizes.
5: An affine transformation is registered and a translation by $(0,4,0)$ is applied to it.
6: The three different modifiers are added to the before created deformer. First the small deformation is applied (see Figure 4.5(b)), then the geometry is transformed by the affine transformation (see Figure 4.5(c)), and finally the second, larger deformation is applied to the geometry (see Figure 4.5(d)). The final result is shown in Figure 4.5(e).

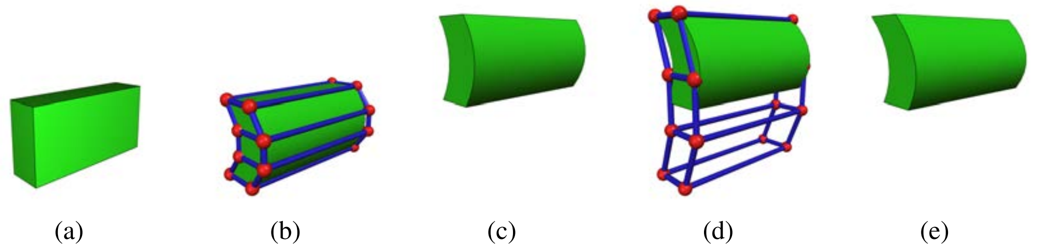


Fig. 4.5 Illustration of the separate steps of the code example in this section. A convex polyhedron (a) is first deformed with a deformation (b), whose dimensions match the axis-aligned bounding box of the polyhedron. Afterwards the deformed geometry is translated upwards (c) and deformed with another deformation with larger dimensions (d). Application of all these modification steps yields image (e).

Procedural Modeling with Scene Graphs in GML

The scene graph package OpenSG [RVN13] has been integrated into the GML in the context of the Master's thesis of Hecher [Hec12]. With the help of this resource, scene graphs (see Section 3.6) can be described within the GML. The important extension is introduction of a new node core type: the *GML context*. With this, standalone GML scripts can be placed within a node of the scene graph and are evaluated at the position of this specific node. This enables the positioning of separate GML models within a scene without the need to adapt code.

Scene Graph Traversal Operators in GML. But first to basic operators for building a scene graph. To retrieve the token of the scene graph root node the operator osg-getroot is provided. With a scene graph node token on the stack several informations can be queried. For example osg-parent retrieves the parent node, and osg-child retrieves a child node based on an index.

Nodes can be assigned a name with osg-setname. To find nodes again in the scene graph, it can be traversed in search of this name with the operator osg-find, which takes a node and a name as input. Traversal is started at the specified node and only the sub-graph starting at that node is traversed in search of a node with the assigned name.

Scene Graph Building Operators in GML. As explained before in Section 3.6.2 in OpenSG a scene graph node is composed of a specific core and the node itself. A node together with a core can be created by the osg-corednode operator, which takes a name, which specifies the type of the core, as input. This operator returns a token for the node as well as one for the node core. For a given node, the node core can also be retrieved by osg-getcore. For a given core, a node encapsulating this core can be generated with the operator osg-makenodefor.

Based on the core type different actions are possible. A transformation core, for example, provides specifying and retrieving its transformation information. The operators osg-translate, osg-rotate, and osg-scale are used for specifying this information and for retrieving, the operators osg-gettranslate, osg-getrotate, and osg-getscale are provided. A newly created node can be added to scene graph as a child of an existing node that is already in the scene graph with the osg-addchild operator.

GML Contexts for Procedural Scene Graphs. GML contexts are provided by a separate resource. A standalone GML program can be specified in these contexts. Each context has its own interpreter and manages its own tokens. Therefore, programs from different contexts cannot share any information with each other. An empty context token is created with the command /gmlcontext node-create. A context can be filled with program code using the node-begin and node-end operators. The node-begin operator takes a GML context token from the stack and activates it. All code specified between node-begin and node-end is then interpreted in this context.

To bridge the gap between this resource and the scene graph resource the osg-ctx2core operator is introduced. This operator takes a context token and generates a scene graph node core from it. The previously mentioned osg-makenodefor operator can then be used to create a scene graph node, which can then be inserted into the scene graph.

Modeling Example with Scene Graphs in GML. The usage of the scene graph capabilities that include GML contexts are illustrated in the upcoming example. This example is again discussed in a step by step manner.

```
1: /Transform osg-corednode !transC !transN
   ;transC (10,0,0) osg-translate

2: /gmlcontext node-create !gmlCtx
   ;gmlCtx node-begin
     %code for any GML model
   node-end

3: ;gmlCtx osg-ctx2core !gmlC
   ;gmlC osg-makenodefor !gmlN

4: osg-getroot ;transN osg-addchild
   ;transN ;gmlN osg-addchild
```

1: A transformation node is generated. The osg-corednode operator returns core and node separately. The translation is applied to the core because the core resembles the transformation and the node is only an encapsulation necessary to insert it into the scene graph.
2: First a token for an empty GML context is created, which is then filled with code between node-begin and node-end. The code for the procedural GML model is interpreted in the before created context.
3: To place the GML model in the scene graph, a core needs to be created from the corresponding context with the osg-ctx2core operator. A node for a core is created with the osg-makenodefor operator.
4: All that is left is to integrate the newly created node into the scene graph. The before created transformation node is used to define an offset to an existing node in the scene graph (in this case the root acquired through osg-getroot). With the osg-addchild operator the desired hierarchy can be defined in the scene graph.

4.1.3 Split Grammars for Architectural Models in the GML

Split grammars (see Section 2.3.4) are very successful techniques for describing architectural buildings. This is mostly due to their conceptual simplicity and the suitability of recursive replacement. Rectangular patterns are very common in classical architecture and are easily described using recursive box splits with boxes as non-terminal shapes, as introduced by Wonka et al. [WWSR03]. Through further publications in this field [MWH*06, HHKF10, KK11] different extensions are added to a common core concept, which will be referred to as *box grammar* from now on.

Split grammars are usually realized within a scripting language. Since the necessary data structures can be easily mapped to the GML, the extensions to split grammars presented in the upcoming sections all use GML as underlying scripting language. This section, in particular, will focus on the realization of split grammars within GML done by our research group.

The split grammar extensions presented in the following two sections include a generalization of box-based split grammars to one that utilize convex polyhedra instead in Section 4.2, and an extension to include free-form deformations into the grammar formalism in Section 4.3. In the former it will be discussed how established split grammar operations are generalized to the use of convex polyhedra and what new operations become possible through this generalization. The latter describes a method to describe curved architecture and presents adaptions of established split operations that operate on curved surfaces.

Split Grammar Syntax within the GML

As mentioned before, a replacement rule in a (context-free) shape grammar replaces the shape that corresponds to the label on the left-hand side through one or more shapes that correspond to labels on the right-hand side. In the GML, a split grammar rule is defined with the following syntax:

```
< Non−Terminal_Name {Condition (optional)} {Rule} [Non−Terminal_List] >
```

The Non-Terminal_Name is a unique literal name that corresponds to the non-terminal shape that is replaced by this rule. Followed by the name is an optional condition block, which is realized as an executable array. The rule is only executed if this condition block evaluates to true. If no condition block is present the rule is always executed. The replacement rule is described in the following executable array. The GML code in this executable array can assume that the shape to replace is pushed to the stack beforehand. For the refinement of the input shape several different operations are provided within the framework. The majority of these operations are discussed in the following two Sections 4.2 and 4.3. Finally, a list of new non-terminal shape labels is required. The amount of different labels provided in this array must coincide with the amount of different shapes produced by the refinement rule. Each of this new non-terminal labels is followed by an executable array that can be used to define individual attributes to these newly generated shapes.

An Introductory Example of GML Split Grammars. Rules that refine a shape with the label Box may look like this:

```
< /Box { index 2 gt }
       { 1 LCS.PX op-repeat-size }
       [[ /Part1 { / first  1 def } ]]
>
< /Box
       { [−1 1] LCS.PX op-split-interval }
       [[ /Part1 { / first  0 def } ][ /Part2 {/ first  0 def } ]]
>
```

There are two replacement rules for shapes with the label Box. Rules are checked whether they are applicable or not in the order of their appearance. In this case, only if the first rule cannot be applied the second one is checked. The second rule is a fallback rule because it has no condition statement and can therefore always be applied.

In this example, the first rule is only applied if the corresponding shape with the appropriate label carries an attribute called index, which has a value greater two. If this is true, a repeat operation with size one is applied in x-direction and all elements of the resulting repetition get the new label Part1. Additionally a new attribute called first is defined explicitly for all these shapes. Later rules may reference this attribute. Some attributes may also be defined implicitly such as the index attribute for repetitions produced by repeat operations.

The second rule is only applied if the condition of the first fails. This rule performs a simple subdivide with an interval of [-1 1]. Negative values in this formalism indicate relative measurements and positive values stand for absolute measurements. So in this case the shape is split into two pieces along the x-axis, whereas the right one has width one and the left one fills the rest of the shape. In contrast to the repeat operation before, the amount of resulting shapes is here defined and each one can be assigned an individual label.

Context-Sensitivity for GML Split Grammars. The syntax also facilitates a way to deal with interconnected structures. Within a rule it is possible to define an arbitrary number of sub-rules, which are executed after the derivation tree starting from the parent rule, which hosts the sub-rules, has finished executing. The sub-rules can be applied to all non-terminal shapes created by the parent rule, which have no associated replacement rule. By applying the sub-rules a new grammar derivation is triggered. The syntax of the sub-rules is a little bit different. Non-context-free rules can be specified by stating one or more non-terminal names. The replacement rule then takes more inputs accordingly.

A simple example looks like the following. An initial Start rule divides a shape through several rule applications into shapes with label Part1 and Part2. After rule derivation of the Start has finished, the sub-rule registered in Start is executed. This rule takes two shapes from which one carries the label Part1 and the other the label Part2. To avoid unexpected behavior the amount of parts with label Part1 and Part2 should be the same. By executing the sub-rule new shapes with label NewPart are generated, which can then be further processed in the usual way.

```
< /Start
        { 1 LCS.PX op-repeat-size }
        [[ /Parts { } ]]

        < /Part1 /Part2
          { ... }
          [ [ /NewPart { } ] ]
        >
>

< /Parts
        { [−1 1] LCS.PX op-split-interval }
        [[ /Part1 { } ][ /Part2 { } ]]
>

< /NewPart
        { op- fill }
        [ ]
>
```

Alternatively, by adding a ∗-suffix to the non-terminal name in the sub-rule, instead of single elements whole arrays can be processed. A sub-rule like the following would take two arrays as input. The first array consists of all shapes with label Part1 and the second consists of all shapes with label Part2. Thus, instead of executing the sub-rule several times for pairs of shapes, this way the rule is only executed one time dealing with all shapes at once.

```
< /Part1∗ /Part2∗
  { ... }
  [ [ /NewPart { } ] ]
>
```

A Compact Notation for Split Grammars. To provide a compact notation for split grammar rules throughout this thesis, the notation from Müller et al. [MWH*06] has been adapted. The introductory example from before translates to this notation in the following way:

$$\begin{array}{lll}
\mathsf{Box} & \rightsquigarrow & \{\mathsf{index} > 2\} \\
 & & \textbf{Repeat}(\mathsf{X}, \mathsf{A}*,\ 1) \\
 & & \{\mathsf{Part1}\} \\
\mathsf{Box} & \rightsquigarrow & \textbf{Subdivide}(\mathsf{X},\ 1\mathrm{r},\ 1) \\
 & & \{\mathsf{Part1},\ \mathsf{Part2}\}
\end{array}$$

To incorporate different variations of repetitions in this notation, the `repeat` operation uses a regular expression to determine the interpretation of the split. The usual `repeat` operation that splits a shape in parts of same size uses the regular expression $\mathsf{A}*$ as parameter. Another conceivable interpretation is a split in parts of alternating size. Such a split has $\mathsf{A}(\mathsf{BA})*$ as the regular expression and needs – besides the sizes for the separate parts – a further parameter to determine among which group the remaining space of the shape is distributed.

Another important difference to notice is that relative size measurements are here indicated by an *r*-suffix instead by a negative value. Furthermore, non-context-free rules that gather shapes together are indicated by a $*$-suffix and are meant to be executed after all shapes carrying the respective labels have been generated.

4.2 Split Grammars on Convex Polyhedra

Box-grammars only feature splitting along the three main axes which span their scope. Splitting boxes only by principal planes, however, is very restrictive. Thus an exploration on what is possible with splits in arbitrary directions naturally let to replacing the box-shaped non-terminal shapes through their generalization: convex polyhedra (see Section 3.5). This section on split grammars on convex polyhedra is based on our work [TKZ*13b] and takes a majority of figures and definitions from this paper. Through this generalization, a more general class of complex shapes is made amenable to the grammar formalism, which in turn enhances its expressiveness. To increase the expressiveness further, additionally to the generalization of the bounding shape, possibly non-convex geometry can be maintained for each shape. This approach only replaces the set of operations on the geometry level, thus it is equally usable with any grammar formalism.

Furthermore, as convex polyhedra are genuinely volumetric, a true mass model can be obtained in contrast to "a collection of paper models" that result from importing meshes. This volumetric representation allows, among other things, to compute volumes and intersections, answer point containment queries, or determine the outer surface.

Split grammars on convex polyhedra have been a joint work of Thaller, Krispel and myself. Contributions in this work are hard to attribute because it was a collaboration of the three of us. My contribution was mainly composed of generalizing existing operations and exploring new operations through applied research to make use of the features provided by this generalized shape representation. These operations are, among others, described in this section. Furthermore, my applied research led to all examples presented in the context of this section. Further results and applications of this kind of split grammar are discussed among others throughout Chapter 6.

Motivating Examples for Split Grammars on Convex Polyhedra. As a simple motivating example, which shows the advantages gained from using convex polyhedra as new non-terminal class, consider a picket fence such as the one shown in Figure 4.6(a). This fence is easy to express through box grammars that support split operations such as the aforementioned approaches [WWSR03, MWH*06, HHKF10, KK11]. These split grammar systems all support box non-terminal shapes along with subdivide and repeat rules (see Section 2.3.4). A fence like this can be modeled by the following split grammar rules:

```
Fence     ⇝  Subdivide(X, 0.1, 1r, 0.1)
             {Post, MidPart, Post}
MidPart   ⇝  Subdivide(Y, 1r, 1r)
             {Pickets, HBars}
Pickets   ⇝  Repeat(X, A(BA)∗, B, 0.1, 0.1)
             {void, Picket}
HBars     ⇝  Subdivide(Z, 1r, 0.1, 1r, 0.1, 2r)
             {void, Bar, void, Bar, void}
Picket    ⇝  ...
Post      ⇝  ...
```

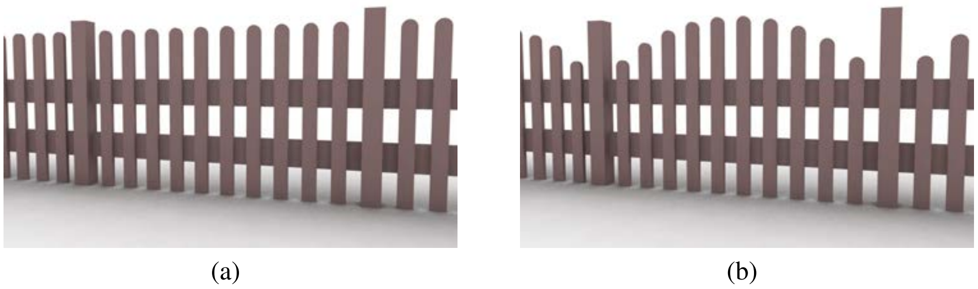


Fig. 4.6 A simple picket fence that can easily be modeled by split grammars based on boxes (a) poses a challenge when it is slightly modified to include a curved boundary for all the single pickets (b). (image source: Thaller et al. [TKZ*13b])

The fence in Figure 4.6(b), however, is slightly different because the space available for each fence segment is no longer shaped like a box. Each fence segment follows a curved boundary and the individual pickets have different height, but still the same rounded shape as before. A sufficiently powerful box grammar can still express this by using a larger bounding box for the fence segment and calculating the bounding curve of the fence within the Picket rule. This way, the bounding box loses its meaning and is no longer denoting the space available to be filled by the shape. Hereby the information of the curved bounding shape is located in two places instead of one: A bounding box is calculated within the MidPart rule and the exact shape based on that bounding box is calculated within the Picket rule. How this rounded fence can be implemented based on convex polyhedra is demonstrated at the end of Section 4.2.4.

The bridge model in Figure 4.7 poses a similar challenge. Again the bridge in Figure 4.7(a) is easy to model using standard box grammars, but the further two bridges are not. The natural bounding shape for most parts of the slanted bridges are not boxes, but sheared ones (parallelepipeds). To achieve all bridges with box grammars, information about the slanting angle has to be propagated to all rules of the grammar. Constructing the bridge like the one in Figure 4.7(a) and then distorting it using a shearing transformation is not an option, as this proce-

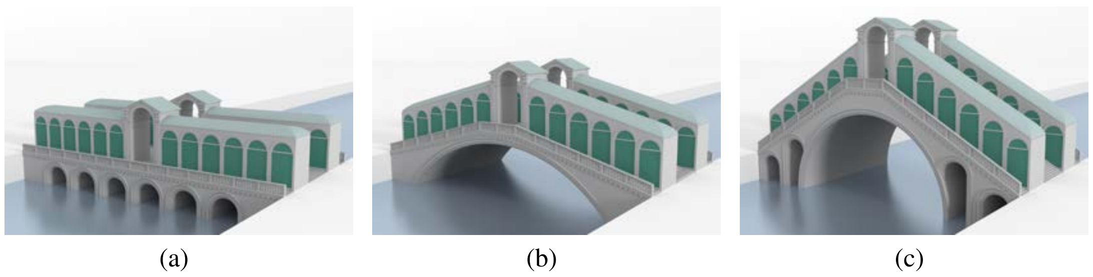


Fig. 4.7 Slanted shapes constitute another challenge for split grammars with box-shaped non-terminals. (image adapted from Thaller et al. [TKZ*13b])

dure would distort various details, most importantly the circular arches, in unacceptable ways. By generalizing box-shaped non-terminals to convex polyhedra these problems are solved. The reconstruction of his particular bridge will be discussed in Section 6.2.2.

4.2.1 Generalization of Non-Terminal Shapes

A convex polyhedron carries more information than a box, but is still simple enough so that rules can still make reasonable assumption about the shape they are replacing. This allows operations to insert sub-shapes that can adapt to the parent shape, which is a convex polyhedron. Similar to how box grammars exploit the simplicity of boxes, grammar rules can exploit the convexity of the bounding shape as a guiding volume for the sub-shapes.

The generalization of the box-shaped non-terminal shape has some impact on the design of the non-terminal shape (see Definition 2.11). The shape S describing the scope, is explicitly changed to a *convex polyhedron*. This affects the local coordinate system too, as it is no longer defined implicitly like in the case of an axis-aligned box as scope. Therefore, a *rigid transformation* is taken to describe the local coordinate system C. The geometry is represented separately from the scope, and does not have to be equal to the scope. In particular, there is no requirement that the geometry has to be convex. Shape operations operate on the scope and the geometry together, but their parameters are usually calculated from only the scope. By using a set of convex polyhedra to represent this geometry, representing non-convex geometry is possible with practically no added implementation cost.

Low-Level Operations Enabled Through Convex Polyhedra. This generalization of the non-terminal shape introduced new opportunities to operate on shapes and on attributes of shapes. As already discussed in Section 3.5, convex polyhedra can be created and modified by adding arbitrary planes. This add-plane operation (which is in this context renamed to: trim operation) allows to trim both the scope and the geometry by an arbitrary plane. Building on this trim operation, the plane-split operation, which encapsulates the main advantage of convex polyhedra, can be introduced. This operation splits a shape with a plane as an input parameter in one part on either side of the plane.

Furthermore, operations such as translate, rotate and mirror that operate on the local coordinate system are allowed too. This is due to the fact that, first, the local coordinate system is no longer fixed and defined implicitly and second, splits in arbitrary directions are permitted.

Finally, in split grammars parameters that are used for operations can either be hard-coded or can be calculated directly from properties of the non-terminal shape. In box grammars such calculated properties would be, for example, the dimensions of the shape. Convex polyhedra, on the other hand, are more expressive, but such values are also harder to compute. To facilitate easy access to parameters that are calculated from the scope of the input shape, a collection of query operations is provided. This collection includes, among others, operations such as the scope-centroid query, which returns the centroid of the current scope, the extreme-point query, which takes a direction and returns the extremal point of the current scope in that direction, and the ray-intersect query, which intersects the current scope with a ray.

Solving Specific Modeling Challenges with Convex Polyhedra. Specific modeling challenges may require direct use of these low-level operations. In case of the winding staircase example shown in Figure 4.8, there is a specific rule that defines the shape of the single steps. This Step rule takes a non-terminal shape, which defines the convex scope for a single step, as input (see Figure 4.9(a)). By trimming this shape with a plane, the lower side of the step is chamfered as seen in Figure 4.9(b). Further rules can tend to the shape afterwards to refine the result (see Figure 4.9(c)).

This does not mean that this specific rule uses hard-coded coordinates to specify this trimming plane. Rather, the extreme-point query operation is used with search directions – specified in local coordinates – to identify characteristic corners of the scope. These are then used for specifying the trimming plane independent from the orientation of the shape. Therefore, the trimming plane, which is calculated from the scope and the corresponding local coordinate system, can adapt to a range of different shapes.

The shapes of the individual steps inherit their outside boundary planes from the scope that describes the entire staircase, which was partitioned into smaller parts for single steps by preceding rules. Consequently, the same rule set for the winding staircase – without any modifications to any of the rules – can be used to build winding staircases in differently-shaped starting shapes as demonstrated in Figure 4.8.

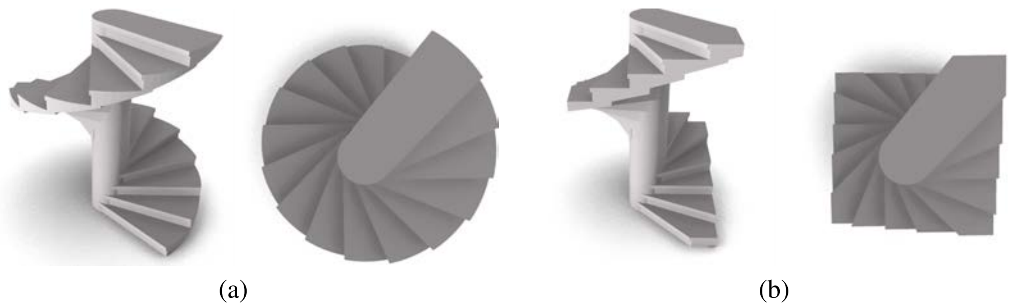


Fig. 4.8 The winding staircase adapts to its surroundings. It can be inserted in any prism-like shape. The shape of the steps is affected by the surrounding scope. The same Step rule is applied independently of the step's shape. A cylindrical scope (a) as well as a box-shaped scope (b) are shown in this example. (image source: Thaller et al. [TKZ*13b])

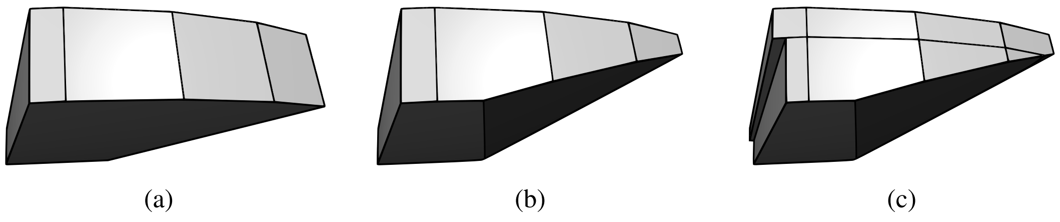


Fig. 4.9 A step for a staircase is modeled by calculating a slanted plane based on the convex scope (a), which is then used to trim the lower side of the step (b). The step is further refined afterwards (c). (image source: Thaller et al. [TKZ*13b])

4.2.2 Generalization of Split Grammar Operations

This section describes the generalization of established shape operations for box grammars to split grammars on convex polyhedra. These include generalizations for subdivide and repeat operations and extrusions. Furthermore, a volumetric counterpart of the component-split from Müller et al. [MWH*06], as well as a non-context-free merge operation, is discussed in this context.

Generalization of Subdivision and Repeat Splits. The subdivide and repeat operations are the foundation of split grammars. In box grammars, the results of these operations automatically adapt to the size of the scope, but these operations can only be applied in the major axis directions of the box-shaped scope. A generalization is necessary to exploit the properties of convex polyhedra and thus enable the application of subdivide and repeat operations in arbitrary directions.

Based on the plane-split, the generalization of the subdivide and repeat operations for convex polyhedra can be defined as:

Definition 4.1 *The* **subdivide** *and* **repeat** *operations take a tuple* $(A, B, \vec{n}, \vec{u}, S)$ *as input, where*

- *A and B are two points that define the limits between which the sizes are measured,*
- $\vec{n}$ *is the normal vector of the splitting planes,*
- $\vec{u}$ *is the direction along which any absolute measurements in S are measured, and*
- *S is specifies the sizes for the parts, which is*
  - *a list of sizes for the* subdivide *operation,*
  - *or a minimum size for the* repeat *operation.*

All parameters are given with respect to the shape's local coordinate system. The operation then proceeds as follows: Planes with normal vector $\vec{n}$ *are constructed through the points A and B (Figure 4.10(b)). The distance between these two planes along the direction* $\vec{u}$ *is calculated. This distance is then used instead of the "length" of the scope to calculate the relative placement of the split planes in the same manner as would be done in a box grammar (Figure 4.10(d)). Finally, the shape is split (Figure 4.10(e) and (f)).*

The fact that $\vec{n}$ and $\vec{u}$ can be specified separately is useful for situations such as the slanted balustrade of the bridge example in Figure 4.12(a). The height of the balustrade, for example, has to be measured along the vertical axis, but for the partition along this vertical direction a slanted splitting plane is used.

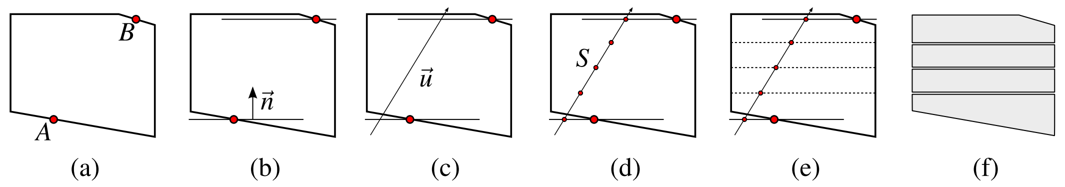


Fig. 4.10 Illustration of subdivide and repeat. The points A and B are supplied as parameters (a). Planes with normal vector $\vec{n}$ are then constructed through these points (b). The distance between these planes along the direction $\vec{u}$ is measured (c) and is partitioned in respect to the size specification S (d). Splitting planes are then positioned (e) and used to split the convex polyhedron (f). (image source: Thaller et al. [TKZ*13b])

The limit points A and B can be calculated in various ways. Three useful methods are provided as pre-defined query operations:

- limits-by-extremes, which determines the farthest points in a given direction $\vec{v}$, i.e. by choosing points A and B such that $\langle A, \vec{v}\rangle$ is minimal and $\langle B, \vec{v}\rangle$ is maximal (Figure 4.11(a)),
- limits-by-rays, which casts rays from the centroid of the scope in a direction $\vec{v}$ and in the opposite direction $-\vec{v}$ (Figure 4.11(b)), and
- limits-by-faces, which chooses two "opposite" faces by casting rays from the centroid and then returns the closest points of those faces along the direction $\vec{v}$ (Figure 4.11(c)).

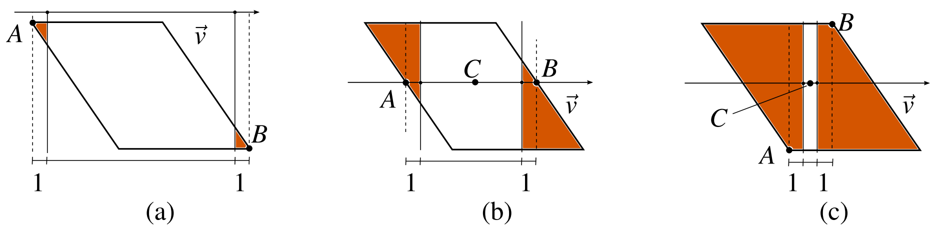


Fig. 4.11 Different ways of choosing the limit points A and B on a split with sizes $(1, 1r, 1)$: the extreme points of the scope along the given direction (a), the points hit by casting rays from the centroid C (b), or the closest points of the faces hit by the rays (c). (image source: Thaller et al. [TKZ*13b])

For axis-parallel splits on boxes all three methods are equivalent. The limits-by-extremes query is the default implementation, which stems from the previous work of Krispel [Kriar]. This method can be understood as building a bounding box along the given direction. The limits-by-faces query, on the other hand, can be used to "rectify" a scope by cutting off a slanted face. Finally, the limits-by-rays query represents a compromise between the other two. Use cases for all three methods are illustrated in Figure 4.12.

These operations make no assumptions about the input shape and yield a defined result for any input shape. However, these rules are expected to be used in rules that make certain assumptions. For example, rules that use the limits-by-face query operation will likely assume that the face found in the search direction is a meaningful feature of the shape, and not a small face that is part of e.g., an approximated cylinder. Within box grammars there are also rules that might assume minimum dimensions or certain proportions for a box-shaped scope, so the situation is no different here.

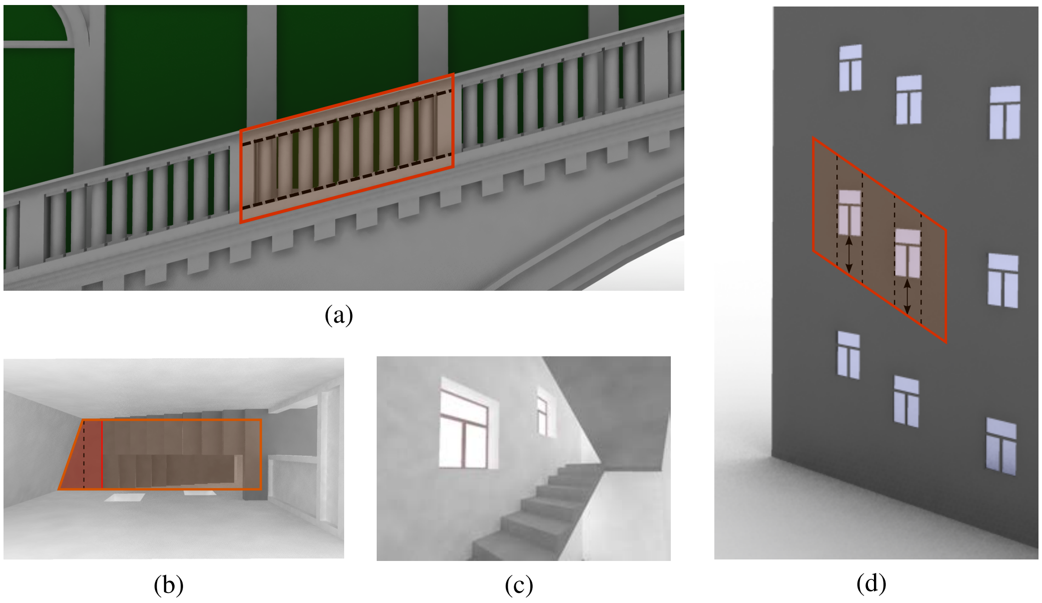


Fig. 4.12 Use cases for the different ways of choosing the limit points for the subdivide and repeat operations. The extreme points in the vertical direction were used to split parallel to the slanted scope's boundaries in the balustrade (a). Ray casting from the centroid of a scope assures that the windows always have the same distance to the stairs ((c) and (d)). Finally, using closest point of the faces hit by the casted ray straightens one side of a scope in a staircase (b) to ensure that there is enough space across the whole width of the landing. (image source: Thaller et al. [TKZ*13b])

Extrusion Operations for Convex Polyhedra. Split operations usually fulfill the containment property, so shapes are replaced by smaller shapes that are contained in the scope of the original shape. In contrast to split operations, extrusion operations create new shapes outside the scope of the original shape. Extrusions allow a more efficient modeling process because details such as moldings can be realized without providing a sufficiently large shape – which has to contain all shapes that would be extruded otherwise – beforehand. Extrusions that are based on convex polyhedra can be realized in two different ways, as introduced by Krispel [Kriar]:

Definition 4.2 *The* **move-plane** *operation takes a plane p that defines a face of the input scope and a distance d as parameters. It constructs a new scope by moving the plane p outwards by the distance d (for an example see Figure 4.13(b)).*

Definition 4.3 *The* **extrude** *operation takes a plane p that defines a face f of the input scope and distance d as parameters. It extrudes the face f to a prism of height d; i.e., it constructs a new shape, whose scope consists of a copy of the given plane p, which is moved outwards by the given distance d, the complement of the original plane p, and of normal planes through the edges of the face f defined by the plane p (for an example see Figure 4.13(c)).*

The Frame-Split. Through the generalization to convex polyhedra an operation that splits a shape into a frame and an inner part can be defined. This so-called frame-split operation, which has been contributed by Krispel [Kriar], is useful in a variety of applications as demonstrated in Figure 4.15.

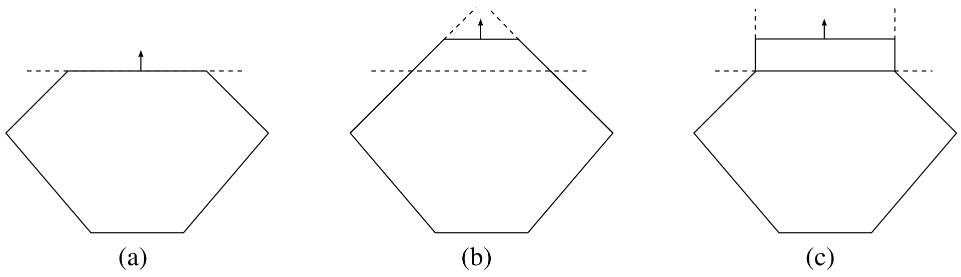


Fig. 4.13 Effect of the move-plane and extrude operations illustrated in two dimensions. Given a convex scope, a plane that defines a face of that scope, and a distance (a), the move-plane operation creates a new shape by modifying the given plane (b), while the extrude operation creates an additional prism shape by sweeping the face over the given distance (c). (image adapted from Thaller et al. [TKZ*13b])

Definition 4.4 *The* **frame-split** *operation takes parameters (F,d), where F is a subset of the planes that define the input scope and d is a distance. The shape is split into one inner part and into several bounding parts.*

The inner part is created by moving the bounding planes of the scope, which are in the set F, along their normal direction by the distance d to create a contracted version of the shape; some bounding planes might not contribute to its surface. A boundary part is created for each face of the inner part. It is bounded by the original scope, by the face of the inner part, and by bisector planes constructed through the edges of the inner part (Figure 4.14(a) through (d)).

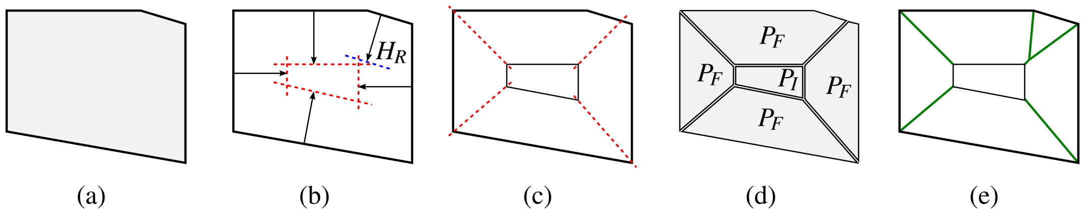


Fig. 4.14 The frame-split is a versatile operation that decomposes a convex shape into a contracted (inner) part and parts for selected bounding planes, which is demonstrated on a two-dimensional example. First, an input polyhedron (a) is shrunk by offsetting its boundary planes (b); note that the plane H_R (blue) does not contribute to the surface of the shrunk polyhedron. For each vertex (edge in three dimensions) of the inner polyhedron, a bisector plane is created (c). The final partition (d) consists of the inner polyhedron P_I and a polyhedron P_F for each edge (face in three dimensions) of P_I which consists of the corresponding edge (face in three dimensions) and its adjacent bisector planes intersected with the original polyhedron. By comparison, offsetting using the straight skeleton (e) produces a different partition. (image source: Thaller et al. [TKZ*13b])

The subset F of the scope planes can easily be constructed using query operations, or can be stored in a user-defined attribute.

This operation is inspired by the component-split from Müller et al. [MWH*06]. In contrast to the frame-split, which operates on the scope, the component-split operates on the geometry and generates new (lower-dimensional) shapes for all of its faces. The frame-split operation yields a non-overlapping subdivision into convex parts of the input scope, hence it is a volumetric counterpart of the component-split operation. The outer parts, as well as the interior part, are available for further refinement.

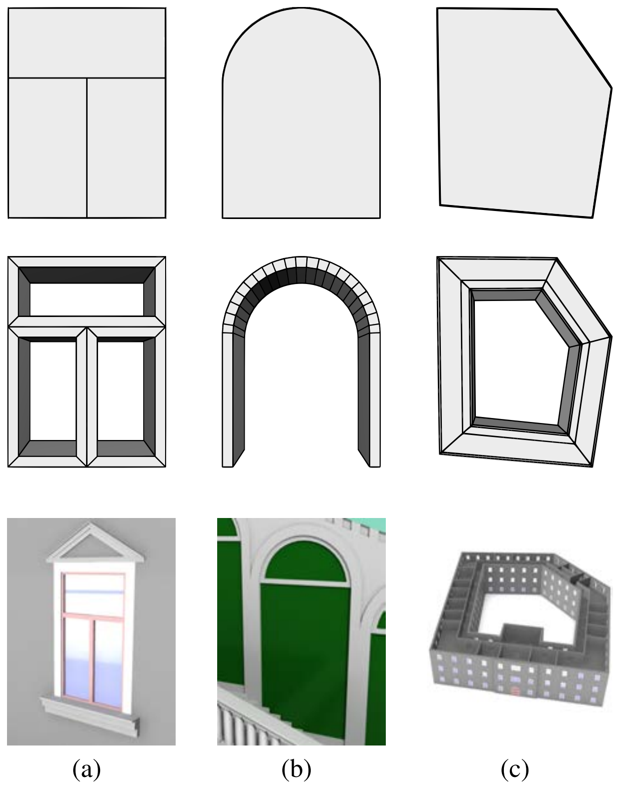


Fig. 4.15 Applications of the frame-split: In the left column (a), a window is created by first splitting a box into three boxes; frame-split on each box (excluding the front and back plane) is used to create the crossbar details (the inner part has been voided). Similarly, frame-split can be used on an arch-shaped scope as shown in the middle row (b), note that the front, back and bottom plane have been excluded. In the right column (b), the frame-split was consecutively applied to partition a convex building into wall, room and hallway sections (c). (image source: Thaller et al. [TKZ*13b])

Merging Shapes. So far all operations have been operating on a single shape only, hence they have been context-free operations. With an extension to the split grammar formalism that allows interconnected structures (like the one described in the work of Krecklau and Kobbelt [KK11]) it becomes possible to implement shape operations that operate on multiple scopes simultaneously. This allows, for example, an operation that joins multiple shapes together into a single shape for further processing.

Definition 4.5 *The* **merge** *operation takes a list of shapes as input and combines them into a single shape. The scope of the resulting shape is the convex hull of all input scopes, and the resulting geometry is the union of all input geometries.*

The merge operation allows procedural creation of non-convex shapes, like L- or U-shapes, from single convex parts. These shapes feature non-convex geometry and a convex scope and are therefore available for further refinement in the same way as entirely convex shapes.

4.2.3 Shape Operations on Shapes with Non-Convex Geometry

Not all useful partitions yield only convex elements. In the following the consequences of allowing non-terminal shapes to have geometry that is not necessarily convex is discussed. This geometry is always independent of the corresponding convex scope.

Operations with Non-Convex Results. Prime example of an operation that generates non-convex geometry is one that inscribes a cylindrical hole into a convex scope. The volume describing the hole is convex, but the remaining volume around it is clearly not convex (see Figure 4.16). It is important how scope and geometry are affected by this operation. For the "inner" result, geometry as well as the scope are both intersected with the same cylinder. For the outer result, however, only for the geometry the cylinder is subtracted. The scope of the outer shape needs to remain convex, so the same difference operation cannot be performed; instead the original scope is maintained. Thus, the round-hole operation splits a shape into two shapes with non-overlapping geometry, but overlapping scopes. Furthermore, this operation can also be provided with its own query operation, which is used to calculate the circle with maximal radius that fits within the two-dimensional silhouette of the scope. An example where this query operation is applied is demonstrated later in Figure 4.19(c).

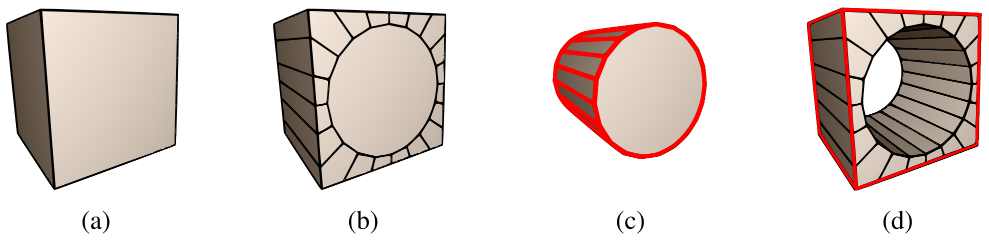


Fig. 4.16 The round-hole operation. In a convex polyhedron (a) a circular hole is cut leaving two results (b): A convex inner part (c) and the non-convex surroundings (d). The red border shows the corresponding convex scope, which is equal to the original scope in case of the non-convex geometry. (image source: Thaller et al. [TKZ*13b])

Splitting Non-Convex Shapes. All split operations are applied to both the scope and the geometry. Consequently, all of these operations can automatically be applied to non-convex geometry as well. After splitting a shape with non-convex geometry, all resulting shapes will still have a convex scope. It is important to notice that even if the scope of the original shape was the convex hull of its geometry (see Figure 4.17(a)), the resulting scopes may be larger than the convex hulls of the geometry of the resulting shapes (see Figure 4.17(b)). With a separate scope-from-geometry operation the scope can be recalculated as the convex hull of the geometry (see Figure 4.17(c)). This step is not done automatically because it is possible that the scope conveys semantic information independent of the geometry. The scope can, for example, specify the original or ideal shape of an object with missing parts. It is desirable that this information is not discarded automatically by split operations. In the example of Figure 4.17 one may want to calculate further split operations based on scope that correlates to the complete cake slice instead on the convex hull of the geometry.

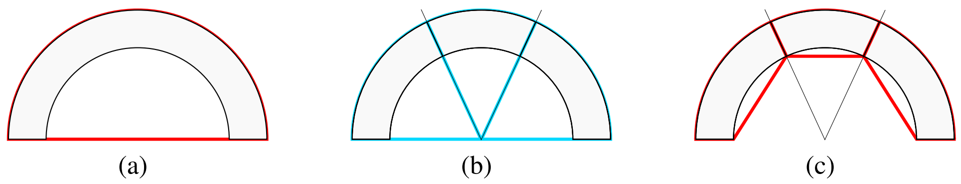


Fig. 4.17 Splitting a shape containing non-convex geometry (a) may lead to shapes with a scope that is larger than the convex hull of the geometry (b). The scope can be recalculated to be the convex hull of the geometry (c). Scopes that are the convex hull of the geometry are shown in red, otherwise in blue. (image source: Thaller et al. [TKZ*13b])

A staircase example is used for demonstrating the application of the scope-from-geometry operation. This time (see Figure 4.18), the starting geometry is L-shaped and is split into volumes for the individual steps of the staircase. The scopes of the shapes are too large after the application of the split and are, therefore, an artifact of the convex hull of the original L-shape. The scopes are, thus, recalculated based on the geometry using the scope-from-geometry operation to insert the Step rule that was discussed before in Section 4.2.1. The result automatically adapts to the space provided by the different scopes that describe the available space for the individual step shapes.

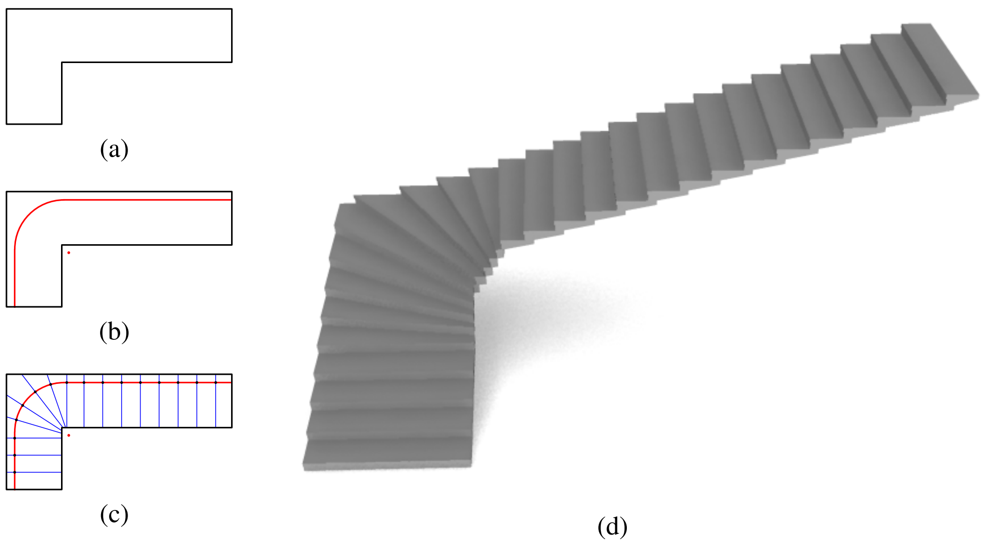


Fig. 4.18 Generation of a staircase in an L-shaped non-convex geometry (a). The step shapes (c) are generated through splits along a polyline (red (b)). To efficiently process the steps further, their scope is recalculated as their convex hull. The Step rule (see Figure 4.9) is then inserted for each step (d). (image source: Thaller et al. [TKZ*13b])

4.2.4 *Special-Purpose Operations for Convex Polyhedra*

The operations that have been discussed so far are universally applicable, but besides them a great variety of special-purpose operations that profit from the generalization to convex polyhedra can be imagined. These operations can be specific to a certain domain (such as architecture) or they can encapsulate useful patterns for certain situations. In contrast to pre-modeled assets, their strengths lie in the re-parametrization and the possibility to be refined further.

Several Examples for Special-Purpose Operations. Examples for useful patterns are the radial-split and the polyline-split. The former divides a shape into smaller cake slices by rotating a plane around a central axis at a given point (see Figure 4.19(a) - (c)), and the latter splits a shape along a series of planes that are placed on a polyline (see Figure 4.19(d)). Distances between these planes are measured along the polyline itself. The polyline-split is expected to be used mostly in situation where the planes do not intersect within the geometry, such as the staircase example in Figure 4.18.

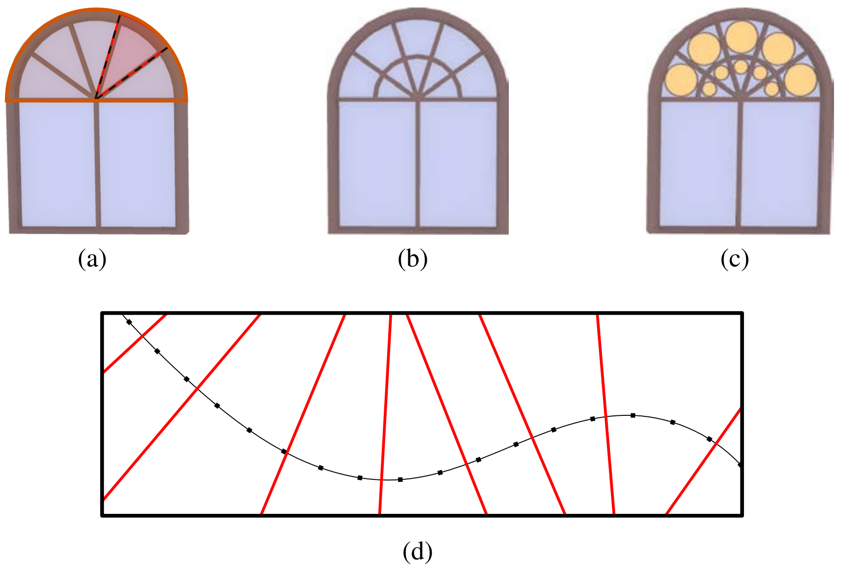


Fig. 4.19 Various round window styles can be created using the radial-split operation: An arch-shaped scope is first split into a lower rectangle and an upper half circle. The half circle is then partitioned into cake-like pieces using the radial-split (a). Using different rules for the cake pieces allows for a great variety of window styles (b), (c). The polyline-split (d) splits a given shape with planes that are placed perpendicular to the given polyline. The planes are placed at specified intervals measured along the polyline. (image source: Thaller et al. [TKZ*13b])

Especially when modeling classical architecture, it is very useful to provide operations for various kinds of arches. Just like the round-hole operation before, the general arch operation splits a given shape into the (convex) volume below an arch and the (non-convex) remaining shape. To maximize the re-usability of this operations two query operations are provided that calculate the exact placement of the arch. These operations (as illustrated in Figure 4.20) maximize the height of the arch such that it still fits within the two-dimensional silhouette (for a provided direction) of a given scope. The query operations that fit these kind of arches into silhouettes of given scopes have been developed by me based on an existing operation by Krispel [Kriar] that takes only the bounding box as guidance.

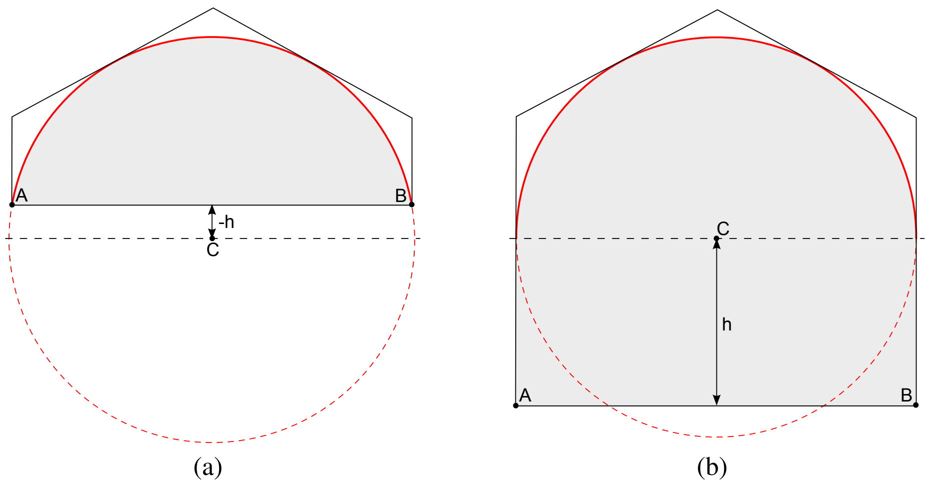


Fig. 4.20 To build an arch between points A and B, a circle segment is fit into the silhouette of the scope. The parameter h is maximized so that the circle segment touches but does not cross the silhouette. For $h < 0$, this leads to a segmental arch (a), while for $h \geq 0$ the result is a round arch (b). In most architectural contexts, it makes sense to restrict the parameter h to one of the two cases. (image source: Thaller et al. [TKZ*13b])

Implementation of the Motivating Fence Example. The arch operation together with the corresponding query operations have been the last puzzle piece missing to create the rounded fence in the motivating example of this section. The challenge in this introductory example was to change the fence from Figure 4.6(a) to have a rounded overall shape as seen in Figure 4.6(b). In a grammar based on convex polyhedra, the overall shape – the scope – has just to be changed to a rounded shape. A single additional rule is necessary to change the rectangular bounding box to the curved shape that is desired. To this rounded shape, the same unmodified, Pickets rule is applied. The following split grammar rules highlight the small changes necessary to achieve the desired result.

Fence	$\rightsquigarrow$	**Subdivide**(**n=u**=X, (A,B)=**extremes**(X), 0.1, 1r, 0.1) {Post, MidPart, Post}
MidPart	$\rightsquigarrow$	**Subdivide**(**n=u**=Y, (A,B)=**extremes**(Y), 1r, 1r) {Boundary, HBars}
Boundary	$\rightsquigarrow$	**Arch**(Y, **calcSegmentalArch**(Y, **h**=0.2)) {Pickets, **void**}
Pickets	$\rightsquigarrow$	**Repeat**(**n=u**=X, (A,B)=**extremes**(X), A(BA)$*$, B, 0.1, 0.1) {**void**, Picket}
Picket	$\rightsquigarrow$	**Arch**(Y, **calcRoundArch**(Y)) {**fill**, **void**}
HBars	$\rightsquigarrow$	**Subdivide**(**n=u**=Z, (A,B)=**extremes**(Z), 1r, 0.1, 1r, 0.1, 2r) {**void**, Bar, **void**, Bar, **void**}

The arch operation is applied to the box-shaped mid-section (see Figure 4.21(a)). The maximal segmental arch (with a given height) is fitted into this shape by the Boundary rule as seen in Figure 4.21(b). The rounded shape is then processed the same way as before. It is split by subdivide and repeat operations, which have been generalized to convex polyhedra, but are applied in the same way as in box grammars (see Figure 4.21(c)). The Picket rule here can be formulated as an arch operation that inserts the maximal round arch into the convex pickets formed by the bounding planes created by the Boundary rule and the Pickets rule (see Figure 4.21(d)). Therefore, there is no need to insert pre-modeled geometry. When the Picket rule is applied, the scope of the Picket non-terminal shape corresponds exactly to that part of the original scope that this particular picket should fit in.

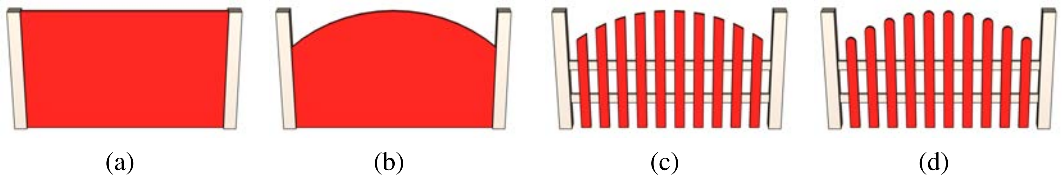


Fig. 4.21 Step by step illustration of the generation of the picket fence with a round boundary. In the box-shaped mid-section (a) the maximal segmental arch (with a given height) is fitted to create the rounded bounding shape for the fence (b). The pickets are then generated using a repeat operation (c). All pickets have a unique, but convex shape due to the round boundary. To complete the fence, round arches are fitted into these convex pickets (d).

4.3 Deformation-Aware Split Grammars

The expressiveness of architectural models acquired through split grammars, as described before, are determined by their concrete hierarchical structuring operations. These operations adapt to planar surfaces, thus models mainly exhibit straight structures or surfaces.

If such architectural models require curved surfaces or curved designs, their creation is very laborious because curved surfaces need to be approximated by planar geometry or appropriately placed pre-modeled parts. Nevertheless, the domain of models that exhibit curved designs and parts is well-suited for a grammatical representation because the hierarchical structure of architectural models is still present.

The typical method to create models that feature curved designs with established procedural methods is first using straight and planar designs to generate a model that is deformed afterwards in a meaningful way using free-form deformations. This approach, however, has two major problems:

- Procedural methods work solely on undeformed geometry. Rules that adapt to available space will not yield desired results because deformations can be used to create more or less space.
- When deformations are applied to hierarchical structures, it is hard to control the the behavior for all individual elements.

This section is based on the on our work [ZTK*13, ZTK*14], which focuses on an extension of the split grammar formalism to integrate free-form deformations into architectural split grammars in a meaningful way. Contribution of this work is mainly attributed to me. Thaller and Krispel contributed through providing the existing shape grammar on convex polyhedra formalism and Edelsbrunner explored the design space of this split grammar by providing some use-case examples. Further results and applications of this extension to split grammars are discussed in Section 6.2.3 and 6.3.3.

Categories of Deformed Architecture. In our work [ZTK*13, ZTK*14] three categories that feature curved parts and designs in architectural models have been identified:

Curved Architecture.
There are many buildings and structures that can be understood as having a straight shape that is distorted to a curved shape (see Figure 4.22) to, for example, adapt to streets or rivers, which do not necessarily follow a straight path.

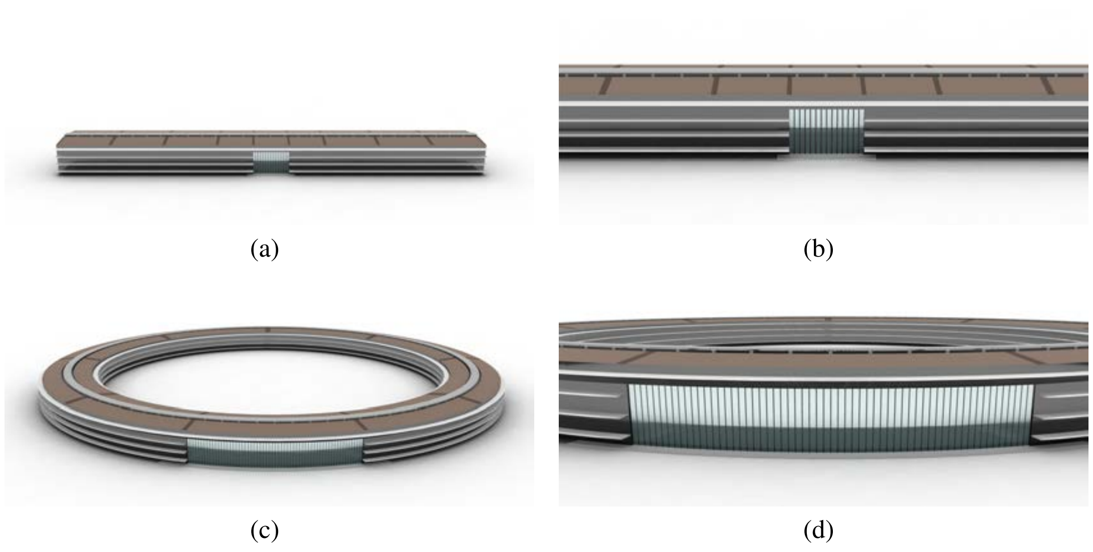


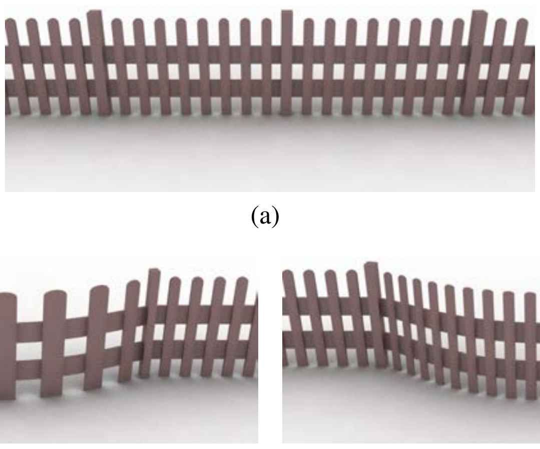
Fig. 4.22 Split grammar rules can be defined and applied to a straight rectangular building (a) with an entrance section made of glass segments (b). Relative and absolute measurements between the building parts are true also after the deformation (c). The number of segments of the entrance section increases in correspondence to the space provided by the deformation process (d). (image adapted from Zmugg et al. [ZTK*13])

Medieval Architecture.
While following the same "straight" ideas as later European architecture, medieval buildings are often slightly deformed due to the limitations of medieval building materials and techniques. This effect is often emulated in modern computer games with a medieval or fantasy theme.

Cartoonish Caricatures of Buildings.
Strongly deformed buildings that do not necessarily need to measure up to statics are at the end of this spectrum. Prime examples here are playful houses from several different cartoons.

To realize buildings out of these three categories, deformations need to be applied at different levels of the hierarchical split grammar refinement process. Thus, deformations can neither be a global post-processing step nor be limited to modeling curved details, but need to be an integral part of the procedural pipeline.

Introductory Examples for Deformation-Aware Split Grammars. An example of a deformed façade can illustrate both aforementioned issues. The number of windows on a deformed façade stays the same, even though more or less space may be introduced by the applied deformation. Furthermore, a deformation of the façade affects all elements; window panes, however, should remain straight and consist of planar surfaces.


(a)

(b) (c)

Fig. 4.23 Illustrative example on how split grammars should adapt to deformations. When a segment of a normal picket fence (a) is deformed the amount of evenly sized pickets should adapt to the deformation (c). Just deforming yields unacceptable results with distorted pickets (b).

Additionally, I will return to the simple example of a picket fence as illustrative example. The following example is just for illustrative purposes and has no real application. A picket fence is composed of equally sized pickets placed at equidistant positions with posts placed in between the individual segments (see Figure 4.23(a)). If one of the segments is deformed all pickets are deformed in this process as shown in Figure 4.23(b). The amount of pickets stays the same even though the segment is stretched and bends to create more space. Due to the deformation the width of pickets also varies based on their position. This result is hardly desirable. The result achieved by deformation-aware split grammars (shown in Figure 4.23(c)) is the most suitable. The width of the pickets stays the same and the amount adapts to the space changes due to the deformation.

4.3.1 Integrating Free-Form Deformations into the Grammar Formalism

Applying deformations should not be a separate post-processing step outside of the actual grammar derivation. This way, grammar operations would be oblivious to the deformations and unable to take them into account. Instead, the grammar system has to operate on deformed shapes. Applying a split operation to a deformed shape has two possible interpretations. A *deformed* split cuts the deformed shape along a plane deformed by the same deformation, while a *straight* split uses an undeformed, straight plane to cut the deformed shape.

For a seamless integration of free-form deformations into a split grammar system, the free-form deformations are added as attributes to the non-terminal shape. The rigid transformation that defines the position of the shape in space is replaced by a list of arbitrary free-form deformations. A list of deformations is used instead of a single one in order to support nested application of free-form deformations as demonstrated in Figure 4.24. Deformations are specified by rules using a deform operation.

Definition 4.6 *The* **deform** *operation takes the parameters (D,C,O) as input, whereas*

- *D is a three-dimensional vector that defines the dimensions (in local coordinates) of the space to deform,*
- *C is a three-dimensional vector (c_x, c_y, c_z) that specifies the amount of control points in x, y and z direction, respectively, and*
- *O is an array of size $c_x \cdot c_y \cdot c_z$, which defines the deformation through specifying all offsets for all control points.*

The shape on which the deform *operation is applied on appends the specified deformation to the list of deformations that define the rigid transformation of that shape. The actual free-form deformation is not performed right away when it is specified; instead, it is postponed until either a straight split is made, or the shape needs to be rendered.*

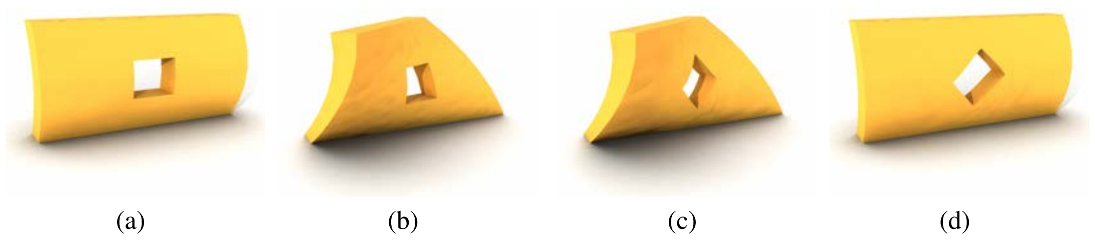


Fig. 4.24 Nested application of free-form deformations in a split grammar. Several deformations are interacting with each other in a simple shape depicting a wall with a window. The initial wall (a) features a slight outwards bending deformation. A further deformation (b) twists this deformed wall. An additional third deformation rotates the window by 45 degrees (c) in the wall, which still features the outwards bending and the twist. The effect of the second twisting deformation can also be reversed while keeping the other two deformations intact (d).

4.3.2 Deformed Splits on Deformed Geometry

Deformed splits are done by planes defined in the local, i.e. undeformed, coordinate space of the shape they operate on. They can thus be implemented very efficiently by applying a classic straight split to the undeformed geometry and annotating the resulting shapes with the same deformations the input shape has. Operations need to take these deformations into account when they are applied; if they do not, the behavior matches the one of a classic split grammar with deformations as a separate post-processing step. From here on this is defined to be the behavior of the standard split operations subdivide and repeat when applied to deformed shapes.

Deformation-Aware Split Operations. In general, lengths of elements are not preserved after applying a deformation. It is desirable that certain elements keep their length independently of the deformation that is applied. Therefore, lengths need to be measured in world space and transformed back into local coordinate space to perform the split operation. This means that splits need to behave differently when applied to deformed geometry. A deformation-aware subdivide operation may yield differently sized parts than the normal operation would generate on the same shape. A deformation-aware repeat operation, on the other hand, may yield more or fewer outputs than the normal operation would produce on this shape because deformations may provide more or less space.

For a normal subdivide operation, a direction and a list of sizes are all the parameters required to divide a shape into parts. The lengths are measured between the extreme points in this direction. However, for a deformation-aware split, the direction is not expressive enough by itself. In general, different parallel lines are deformed differently. One way to resolve this ambiguity is to provide an additional point in the deformed space (see Figure 4.25). This point, together with the direction in which the split should occur, is then used to calculate the distance between the two extremal points in the deformed space.

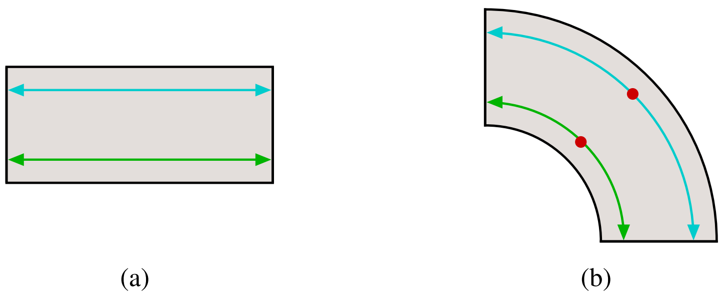
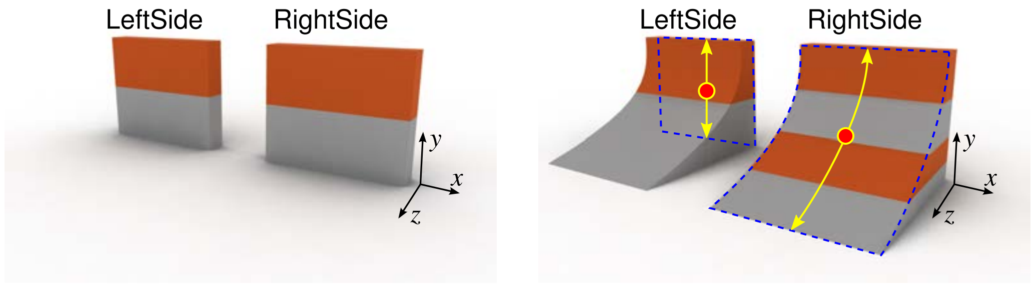


Fig. 4.25 In an undeformed space a direction to measure along is sufficient (a). However, in general, in a deformed space only a single direction is not enough. An additional point (shown in red) within the deformed space is needed to clear this ambiguity (b). (image source: Zmugg et al. [ZTK*13])

This leads to the definition of the deformation-aware subdivideD and repeatD operations.

Definition 4.7 *The deformation-aware* **subdivideD** *and* **repeatD** *operations take an additional point, which is provided in the local coordinate system of the shape to split, as input. The extremal points, which are used to measure distances in between, are calculated based on this point and the given split direction. Distances are measured in the deformed space, which affects the outputs of both operations. The* subdivideD *operation may yield different-sized parts than its regular counterpart and the* repeatD *operations may yield more or less elements than its regular counterpart.*

An Illustrative Example for Deformed Splits. An application of a repeatD operation on two objects with the same deformation is illustrated in Figure 4.26. When applied to undeformed shapes (see Figure 4.26(a)) this operation yields the same result on both parts independent of the point used for measuring, but for deformed shapes (see Figure 4.26(b)) the operation yields two different results, even though it was provided with the same input size. The difference between the two situations is that the additional point for measuring was placed differently. The point is placed by projecting the centroid of the undeformed shape to the back and the front face, respectively. The left box uses a point on the back face for measuring, which leads to a result that is equivalent to that of a normal repeat operation because the back face is not affected by the deformation. The right box, on the other hand, places its point for measuring on the front face, which is affected by the deformation. Thus, the operation uses the additional space which leads to more repeated elements.



(a) (b)

Fig. 4.26 For undeformed shapes (a), the repeatD operation yields the same result on both parts. However, for deformed shapes (b) the same operation yields different results based on the measurement. The additional point for measuring (red) was placed by projecting the centroid of the undeformed shape to the outside boundary face (back and front face respectively). (image source: Zmugg et al. [ZTK*13])

The grammar rules for this example can be written as follows:

$$
\begin{array}{ll}
\mathsf{Box} & \rightsquigarrow \ \textbf{Deform}(\mathsf{localBB},\ (2,4,2),\ [...]) \\
 & \quad \{\mathsf{DeformedBox}\} \\
\mathsf{DeformedBox} & \rightsquigarrow \ \textbf{Subdivide}(\mathsf{X},\ 1r,\ 1,\ 1r) \\
 & \quad \{\mathsf{LeftSide},\ \textbf{void},\ \mathsf{RightSide}\} \\
\mathsf{LeftSide} & \rightsquigarrow \ \textbf{RepeatD}(\mathsf{Y},\ \mathsf{A}*,\ 2,\ \textbf{rayIntersect}(\mathsf{Z})) \\
 & \quad \{\textbf{fill}\{\mathsf{mat} = \textbf{setMaterial}(\mathsf{index}\ \%\ 2)\}\} \\
\mathsf{RightSide} & \rightsquigarrow \ \textbf{RepeatD}(\mathsf{Y},\ \mathsf{A}*,\ 2,\ \textbf{rayIntersect}(-\mathsf{Z})) \\
 & \quad \{\textbf{fill}\{\mathsf{mat} = \textbf{setMaterial}(\mathsf{index}\ \%\ 2)\}\}
\end{array}
$$

The label Box refers to a box-shaped (undeformed) starting shape. The deform operation takes as its input the bounding box (in local coordinates), the number of control points in x, y, and z-axis direction as well as an array of individual offsets for these control points. The actual list of offset vectors has been omitted for brevity. Utility functions can be defined to allow more convenient specification of common deformations. After setting the deformation, one can either use standard split operations (subdivide in this case) or the new deformation-aware operations, which are indicated by a D-suffix. The new operations need an additional point as input, which is in this example calculated by casting a ray from the centroid and intersecting it with the shape. Finally the fill operation renders the shape with the material set to the attribute called mat.

A Façade Example using Deformed Splits. Figure 4.27 demonstrates the effect of deformation-aware operations in an example of a façade. Splits along the width of the façade are done with respect to the deformation. Note that more windows are placed on the façade using the additional space that was generated through the deformation. All windows on the deformed façade have the same width. To cope with the additional space provided by the deformation, more splits are introduced in the local coordinate space (see Figure 4.27(c)). Note that the windows in the local coordinate space have varying widths which will be transformed to uniform lengths by the deformation.

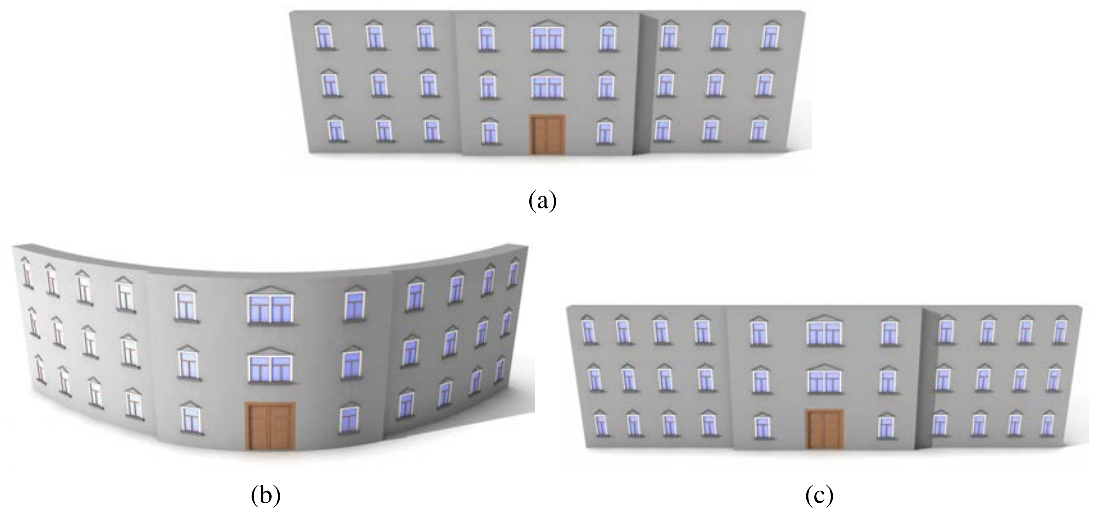


Fig. 4.27 Applying deformations to a straight façade that is defined using a split grammar (a) yields a different number of windows on the side parts (b) when carried out with deformation-aware operations. In local coordinate (undeformed) space (c) more splits need to be introduced to cope with the additional space provided by the applied deformation. (image source: Zmugg et al. [ZTK*13])

4.3.3 Straight Splits on Deformed Geometry

There are also situations in which deformed geometry needs to be split along straight, undeformed planes, i.e., planes given in world coordinates. Window panes, for example, should not usually follow the curvature of a wall, but should be planar (see Figure 4.28).

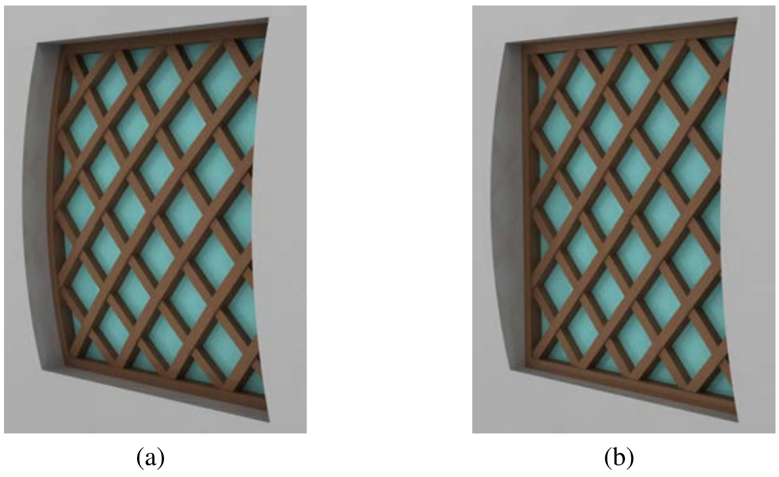


Fig. 4.28 When a façade is deformed as a whole, the windows are deformed as well (a). Straight splits can be used to insert straight windows (b) in deformed walls. (image source: Zmugg et al. [ZTK*14])

To address this issue, a new grammar operation called bake is introduced.

Definition 4.8 *The* **bake** *operation takes one integer x as input, which defines the number of innermost deformations of a non-terminal shape that are applied to the shape's actual geometry and scope. This certain deformations are then removed from the list of deformations. The resulting shape is visually equivalent to the shape on which the operation was applied, but the planes used by subsequent splitting operation will no longer be affected by the baked deformations.*

Unfortunately, applying a free-form deformation to the scope rarely yields a shape that fulfills the requirements imposed by the grammar system, meaning in general it is neither a box nor a convex polyhedron any more. The shape resulting from this bake operation can therefore only be an approximation of the ideal deformed shape; its scope, in particular, will have to be a bounding shape of the deformed scope.

For a box grammar this is a serious problem. An axis-parallel bounding box can be a very poor approximation of the deformed shape and the coordinate axes will no longer have a special meaning for the shape, so split operations, which only operate on these axes, will hardly be applied as desired. Thus, it is necessary to use at least a grammar based on convex polyhedra (see Section 4.2) to efficiently use this feature. Through a convex polyhedra-based representation the deformed scope can be approximated by its convex hull, and then be split in arbitrary directions. This allows to specify splitting planes in the deformed coordinate space, which is – for the before mentioned use case of planar window panes – necessary to determine the correct orientation of the window pane according to the wall's deformation.

Different Use Cases for the Bake Operation. Sometimes it is not necessary to approximate the deformed geometry at all. It most cases, like the window pane example, it is sufficient to use the approximated scope as geometry, as this example only relies on the shape of the deformed scope. However, in cases where the convex hull does not describe the shape accurately enough for further processing, the geometry itself needs to be approximated too. An example for such a case will be discussed in Section 6.2.3.

Implementation Details of the Bake Operation. To implement the bake operation, deformations need to be applied to scope and geometry of a shape. The former is achieved by calculating the convex hull of a set of deformed sample points, which are placed uniformly on the undeformed geometry. However, deforming arbitrary geometry, represented by sets of convex polyhedra, is a much harder problem, as the result again has to be represented as a set of non-overlapping convex polyhedra for further processing. Fortunately, the geometry approximation problem can be solved simply and efficiently for a common class of deformations.

Consider a deformation with a grid of $n \times 2 \times 2$ control points applied to a convex shape. It is further required that for each index i, the control points $C_{i,0,0}$, $C_{i,0,1}$, $C_{i,1,0}$ and $C_{i,1,1}$ are coplanar. It follows that any points that share the same x coordinate are transformed to coplanar points by this deformation. The deformed shape can therefore be approximated by slicing the undeformed geometry into smaller polyhedral pieces along the x axis, and then calculating the convex hulls of the deformed vertices of each piece.

4.3.4 Deformations of Adjacent Objects

When modeling a building, it is useful to specify the deformations for individual walls and roof segments separately. This has several advantages over using a single global deformation for the entire building. On the one hand, for non-rectangular buildings, separate deformations are significantly easier to specify and, on the other hand, in comparison to separate deformations a single deformation would require a massive amount of control points, which increases complexity and computation time.

The effect of applying a single global deformation versus separate single deformations is demonstrated in an example of a simple house (see Figure 4.29). For the global deformation all façade elements, like windows, are affected by this deformation. In contrast, by using separate, local deformations the deformation in the tangent plane of the surface is controllable independently from the neighboring walls. Here the façade elements are not influenced by the deformations of adjacent walls. To highlight this effect further, an abstract example that features straight lines on the walls is provided in Figure 4.29 as well. A global deformation affects and deforms these lines in the tangent plane, but with separate deformations these lines remain unaffected by deformations of neighboring walls.

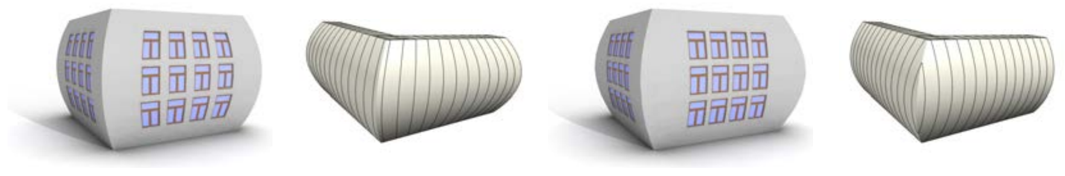

(a) (b)

Fig. 4.29 The walls of a house are deformed by bending each wall in outward direction. When a single global free-form deformation is applied to this model (a), straight structures on one wall, like windows in this case, are influenced by the outward bending deformation of adjacent walls. By deforming each wall separately this effect can be controlled. The windows on the right (b) remain straight in vertical direction and are not deformed at all in the tangent plane. (image adapted from Zmugg et al. [ZTK*13])

Connecting Deformed Adjacent Parts. However, deformations of spatially neighboring parts may lead to problems on the boundaries between these parts, as indicated in Figure 4.30. Even though there are no issues in the undeformed case (see Figure 4.30(a)), depending on the deformation, geometry will be missing (see Figure 4.30(b)) or there will be artifacts from overlapping geometry (see Figure 4.30(c)). Adjacent walls and roof segments thus have to be connected properly. In this scenario appropriate Boolean operations are used to generate the correct corner geometry between two separately deformed parts. While it is conceivable to specify the necessary operations by hand as part of a specific grammar, connecting parts adds a lot of complexity to even the simplest grammar specifications. Note that this requires extensions beyond context-free grammars, like the extensions presented by Krecklau and Kobbelt [KK11].

A solution for this problem is offered by the observation that a large class of conventional buildings can be covered by a few special-purpose operations and non-terminal shape classes in the split grammar system. These operations are:

- Replace Ground Polygon by Line Segments,
- Replace Line Segments by Wall, and
- Replace Line Segments by Wall and attached tilted Roof Segment.

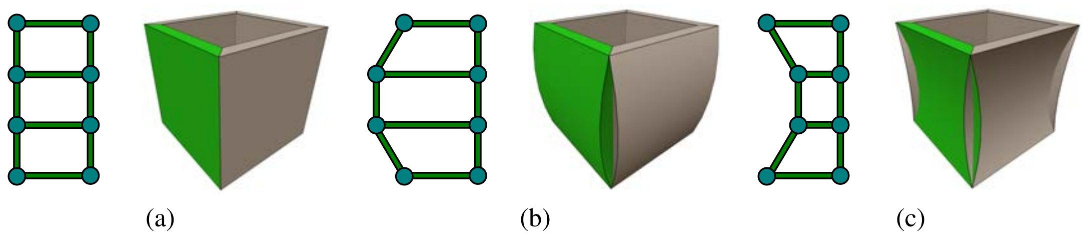


Fig. 4.30 Adjacent local deformations of walls may lead to artifacts. Starting from an undeformed wall setup (a) different deformations are applied. Depending on the deformation it can happen that there is missing (b) or surplus (c) geometry. Boolean operations are used to overcome these issues and generate the correct corner geometry between two deformed walls. (image source: Zmugg et al. [ZTK*13])

Thus, a building can be specified as a polygon whose sides are annotated with attributes and informations about the walls and roofs. Through these operations and non-terminal shape classes modeling a building integrates more seamlessly into the grammar setting as it allows separate rules to define the behavior of different sides of the building.

Furthermore, when these operations are used, the system can automatically keep track of all adjacencies between the walls, and the adjacencies between the walls and the corresponding roof segments. The remaining adjacency information – for the neighborhood between the roof segments – can be by solving the weighted straight skeleton problem [AAAG95, EE98] for the roof. Knowing the complete adjacency graph between the wall and roof segments allows the system to implicitly specify the Boolean operations necessary to properly connect the separately deformed parts.

Obtaining the Correct Corner Geometry. To obtain the correct corner geometry through Boolean operations, the initial geometry of walls and roof segments is first extended in length and height to infinity. On either side of the wall, a half space that occupies all space that is not taken by the extended geometry is generated. These *minus spaces* will later be used to trim the adjacent parts. Roofs also generate two different minus spaces, one above and one below the actual roof geometry. The extended geometry can be further refined using split grammar rules. These rules refer to the original bounding shape not the one of the extended geometry.

After rule processing has finished, deformations are calculated automatically; finally, the deformed geometry generated for the walls and roof segments is trimmed by the deformed minus spaces of the neighboring walls and roofs (see Figure 4.31). For roofs and walls alike, the outer minus space is used for convex corners, and the inner minus space is used for reflex corners. The resulting corner geometry remains correct even in the presence of completely independent deformations applied to the individual walls and roof segments. To achieve ledges – extrusions in the wall shape – that run across a corner, an additional Boolean operation is necessary that intersects both extended wall geometries with each other.

Let us call the extended parts of a wall A, $\mathsf{A_l}$ (left side) and $\mathsf{A_r}$ (right side). The corresponding minus spaces placed on the exterior and on the interior of the wall are called $\mathsf{A_{e-}}$ and $\mathsf{A_{i-}}$, respectively. The Boolean expression for the convex corner geometry $\mathsf{c_{AB}}$ between walls A and B is the following:

$$\mathsf{c_{AB}} = (\mathsf{A_r} \cap \mathsf{B_l}) + (\mathsf{A_r} - \mathsf{B_{e-}}) + (\mathsf{B_l} - \mathsf{A_{e-}}).$$

In case of a reflex corner, the opposite minus spaces A_{i-} and B_{i-} are necessary. The two difference parts of the Boolean expression maintain the correct shape of the corner, while the intersection part preserves extrusions across corners, such as ledges.

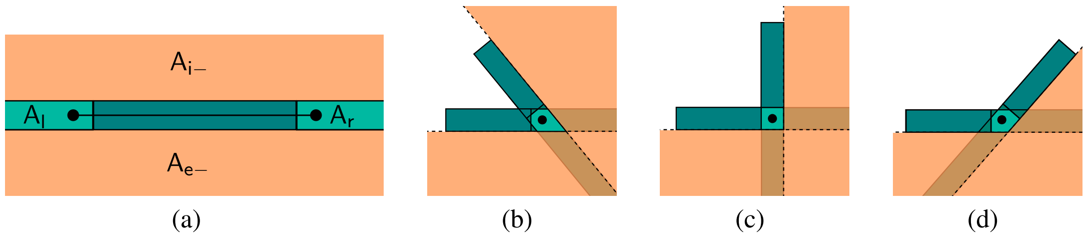


Fig. 4.31 Illustration of the additional parts created for a wall between two endpoints (a). For a wall A (green) (a) the prolonged parts (blue) are called A_l for the left side and A_r for the right side. The minus spaces (orange) placed on the exterior and on the interior of the wall are called A_{e-} and A_{i-}, respectively. When two neighboring walls meet at different angles ($< 90°$ (b), $= 90°$ (c), $> 90°$ (d)) the minus spaces trim all superfluous parts. In this illustration only convex corners are depicted; for reflex corners the opposite minus space is used. (image adapted from Zmugg et al. [ZTK*13])

Implementation of the Minus Spaces. In practice, the minus spaces and the extended walls necessary for Boolean operations are represented by sufficiently large boxes in order to avoid having to deal with infinite shapes (see Figure 4.32). As a performance optimization, each wall is split into central and peripheral parts before the Boolean operations are applied. These peripheral parts include the extension as well as the part of the wall geometry near the boundary of the original wall shape. This is where the intersection with the adjacent walls can be expected to happen. The central part, which usually contains most detail, does not need to be trimmed by neighboring walls; only the peripheral parts take part in the Boolean operations. All parts, however, are trimmed by the minus spaces of adjacent roof segments.

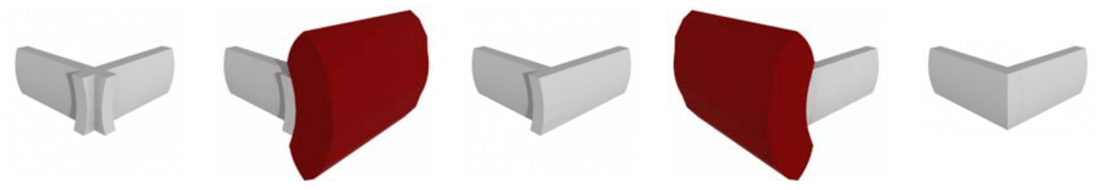


Fig. 4.32 The minus space, approximated by a sufficiently large red box, is deformed in the same way as the corresponding wall and trims the surplus geometry of the neighboring wall. (image adapted from Zmugg et al. [ZTK*13])

4.4 The GML Compositor

Creating three-dimensional content is no easy task. Usually this task is very time consuming and requires expert knowledge. As elaborated before, procedural modeling can be used to ease this process. Especially for the creating of man-made shapes procedural descriptions are perceived to be the most appropriate because such shapes exhibit a great number of regularities and similarities due to a number of reasons. These range from functionality over manufacturability to aesthetics and style factors.

However, for the creating of procedural descriptions some kind of scripting language is always required. A text editor for a scripting language is the usual user interface for procedural three-dimensional content creation. Such a code editor is, however, not accepted by the most artists because only few of them are good programmers. Data flow graph based visual programming languages like in Houdini [Sid13] or Grasshopper [Rob13a] have emerged to tackle this problem. These languages feature a data flow graph, which creates the necessary code in the background, to represent the three-dimensional model. By connecting input and output ports of nodes in this data flow graph, creating code can – to the most part – be avoided. Another similar approach was introduced by Patow [Pat12], who introduced ways to edit shape grammars through the underlying graph. The drawback of these methods is that such graph-based languages are not necessarily easier to understand than a normal textual representation, as Green and Petre discussed in their work [GP92].

The GML Compositor is a system developed by our research group. My contribution in the development was mainly composed of further extending the system and adapting new modeling vocabularies to the underlying formalism, which itself had to undergo changes to support further modeling domains. These new domains include, among others, scene graphs and deformed convex polyhedra.

The GML Compositor features direct manipulation of the procedural description on the concrete three-dimensional model, while retaining the expressiveness of data flow graph-based methods. To use this system no knowledge of the underlying representation – the code – is necessary. Data flow graphs tend to grow very large and complex already for medium sized models. To avoid confusion, the graph, as well as the generated GML code, remains hidden from the user in the GML Compositor. A screenshot of the user interface is seen in Figure 4.33.

Based on the work of Thaller et al. [TKHF12, TKZ*13a] this section presents the theoretical foundations – the use of code graphs – of the GML Compositor. Additionally, common modeling domains and their realization within the GML Compositor are described together with the most important of the corresponding supported modeling operations. The examples discussed in this section have been realized using direct manipulation on a visible model only. The underlying code graph was never visualized during the modeling process. These examples and applications were also featured in our work [TKHF12, TKZ*13a, ZKT*14, ZTH*12].

4.4.1 Data Flow Management

To realize procedural interactive modeling editors it is necessary to describe the dependencies and relations that are present in the modeled scene. An appropriate representation allows changing of parameters and re-evaluating of the model.

Within the procedural modeling editors Houdini [Sid13] and Grasshopper [Rob13a] data flow graphs are used to visualize the modeling history and dependencies throughout the model. Each modeling operation is realized as a node in that graph that can be attached interactively in the system. A slightly different graph concept, namely code graphs, have been utilized in the the GML Compositor by Thaller [Thaar]. Both of these graph concepts are explained in short in the following.

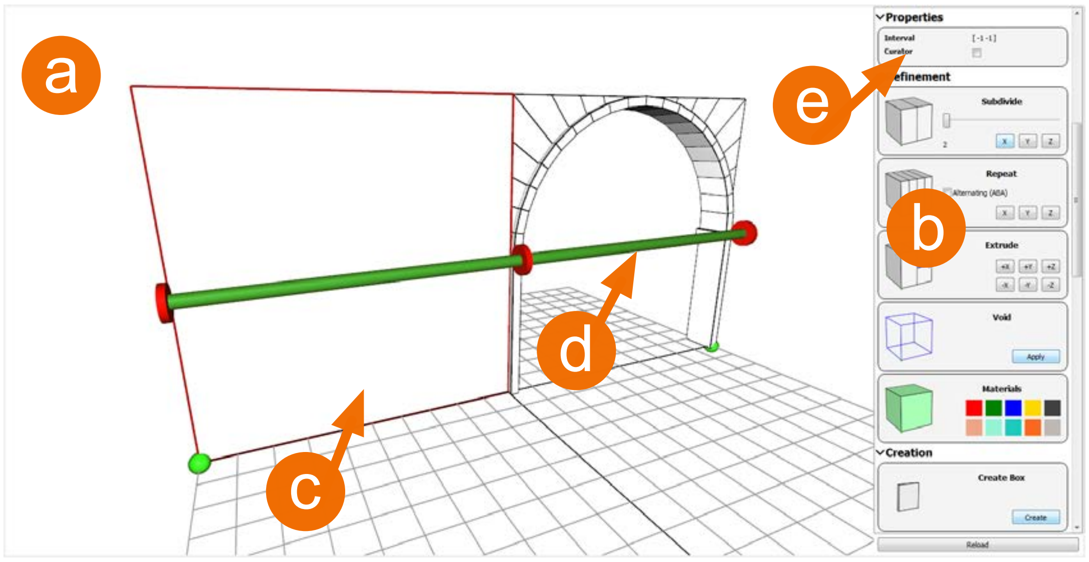


Fig. 4.33 Graphical user interface of the GML Compositor, which includes the three-dimensional viewer on the left side (a) and a list of available operations on the right side (b). In the viewer a simple split grammar model is edited interactively. The selected component (c) is highlighted by a red border and the last applied split operation to this shape is highlighted by the three-dimensional manipulator (d). The parameters of this operation can be changed interactively by dragging the middle disc of the manipulator, or by specifying the sizes manually via the properties menu (e).

Data Flow Graph Basics

A data flow graph represents dependencies between inputs and outputs between a number of different operations. Data flow graphs are data-driven and are used to simulate data dependencies within programs. They are an essential tool used for compiler optimizations and static program analysis.

Definition 4.9 *A **data flow graph** $G = (V, E)$ is a directed graph, where*

- *V is a set of nodes representing operations. Each operation has a set of **input** and **output ports**.*
- *E is a set of edges (acres) connecting input and output ports of nodes and represent the data flow within the graph.*

Nodes consume data from input ports and produce data, which is provided to its output ports. The operation the node represents can be executed at the moment when all input ports have their data available. Many nodes can be ready for execution at the same time.

Each node, which can either represent a single function, whole sub-system, or just a constant value, is represented as a box within this graph. The activity of each node depends on all the nodes which are connected to its input ports, so each functional block may has to wait until a sufficient amount of data is provided before it can process the input. An example for a program solving a quadratic equation is presented in Figure 4.34.

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$$

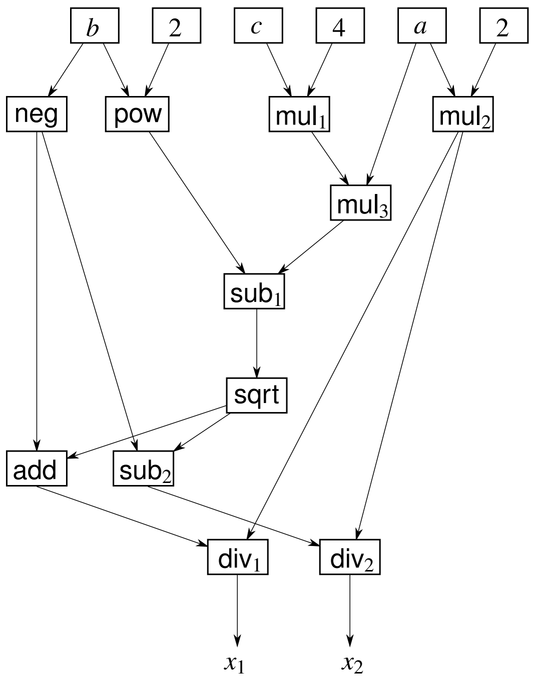


Fig. 4.34 The solutions for a quadratic equation $a \cdot x^2 + b \cdot x + c = 0$ computed with a data flow graph. The solution formula is shown at the top of the figure. Nodes in the graph represent operations and edges (acres) connect input and output ports of those operations. Constants are realized as operations with constant output and, thus, no input ports. An operation can be executed to provide the output ports with values. This happens if all input values have been computed and are available at the input ports.

Code Graphs for the GML Compositor

A code graph is a *hypergraph* consisting of nodes and *hyperedges*. Strictly speaking, a code graph is a *term graph*, which features a terminology that is different to those of a data flow graph. In a data flow graph, nodes are labeled with operations, and they are connected with edges, which transport values through the graph. In a term graph (see Figure 4.35), however, values are stored in nodes, which are labeled with a type and represent graphical objects. Hyperedges are labeled with operations or literal constants, and connect two sets of nodes, which may have different sizes. The following code graph definition is reused from Kahl et al. [KAC06]:

Definition 4.10 *A **code graph** over an edge label set* ELab *and a set of types* NType *is defined as a tuple* $G = (\mathcal{N}, \mathcal{E}, \mathsf{In}, \mathsf{Out}, \mathsf{src}, \mathsf{trg}, \mathsf{nType}, \mathsf{eLab})$ *that consists of:*

- *a set* $\mathcal{N}$ *of **nodes** and a set* $\mathcal{E}$ *of **hyperedges** (or **edges**),*
- *two node sequences* In, Out*:* $\mathcal{N}^*$ *containing the **input nodes** and **output nodes** of the code graph,*
- *two functions* src, trg*:* $\mathcal{E} \to \mathcal{N}^*$ *assigning each edge the sequence of its **source nodes** and **target nodes** respectively,*
- *a function* nType*:* $\mathcal{N} \to$ NType *assigning each node its **type**, and*
- *a function* eLab*:* $\mathcal{E} \to$ ELab *assigning each edge its **edge label**, representing the corresponding operation.*

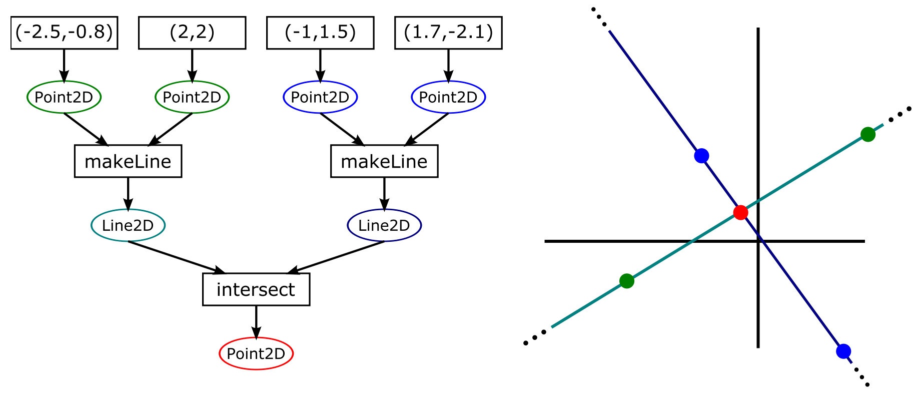


Fig. 4.35 A code graph is a hypergraph consisting of nodes, which correspond to (intermediate) values and graphical objects, and hyperedges, which represent the operations applied. Nodes are represented as ellipses and hyperedges are visualized as boxes. This example shows a code graph that carries out a simple construction: Two points define a straight line; two lines yield an intersection point. The colored points on the right correspond to the colored nodes in the code graph (left). (image source: Thaller et al. [TKHF12])

Hyperedges can have any number of source and target nodes. Hyperedges with no source nodes correspond to constants. Additionally, code graphs (in the context of the GML Compositor) are required to be acyclic.

4.4.2 Executing Code Graphs

Executing a code graph means executing all of its edges, which produce values for their target nodes. The order in which the edges are executed is only restricted by the data dependencies, which are explicitly represented in the code graph. So an edge can only be executed when all its source nodes provide their values. Based on this constraint, the execution order can be chosen arbitrarily; independent operations can be evaluated at any time.

The ability to reorder the execution of the edges is an essential property to support direct manipulation graphical user interfaces. When a user edits parameters or parts of a model, only the operations that are dependent on this changed data need to be re-evaluated; only a part of the code graph is affected, so only this part is executed anew with changed values. A strict linear execution order would imply that the entire model is re-evaluated for each change, even if it's just a small local adjustment.

This section will explain in the following the concepts necessary to realize the GML Compositor. These concepts have been presented in the work of Thaller et al. [TKHF12, TKZ*13a] and the design and implementation of those is – to the most part – contributed to Thaller [Thaar]. I was involved in designing the concept of incremental updates in combination with scene graphs.

Implementation of Repetitions in the GML Compositor

Code graphs, as used within the GML Compositor, provide no facilities for loops. Basic code graphs can be extended to support repetitions in various ways. In the GML Compositor it is important that cases of repetition that arise in procedural modeling tasks are covered, which include nested repetitions as well. Furthermore, it is desirable to avoid any explicit looping constructs within the code graph; operations should be repeated implicitly when applied to a collection of objects. Lastly, it is important to avoid non-termination and undecidability issues and thus Turing-completeness.

Existing procedural modeling systems based on data flow graphs use a concept in which a linear stream of tokens is transported by edges. Tokens all get treated the same by subsequent operations, without support for nested structures. Within direct manipulation interfaces a linear token stream is not suited. Imagine a model of a building façade, which consists of several stories, each of which contains several identical windows, which again consist of a collection of separate window panes. A user will zoom in to one window for editing purposes. Operations should behave consistently, independent of whether a model with a single window or a collection of windows is edited. Either way, the system needs to remember that a collection of window panes belong to a window. When all window panes are transported in a linear stream of tokens they loose all connection to their respective windows.

The basic idea behind the repetition concept of the GML Compositor is the following: First, the set of available types is extended by list types. Therefore, for every type $t \in \mathsf{NType}$ there is a type $\mathsf{List}[t] \in \mathsf{NType}$. This definition also allows nested lists of arbitrary depth. Second, operations are automatically mapped over lists, which means when an operation is applied to a list, it is applied to all elements of the list and the resulting values are gathered in an output list.

Operations also need to produce meaningful results when all their inputs or only a subset of the inputs are of a list type. In case of more lists, the operation is applied to the corresponding elements of the lists. If one value is not of a list type, then this value is used throughout all computations with the changing list elements. It is, of course, assumed that lists have been arranged properly. For example take the makeSegment operation which connects two points. This operation reacts accordingly whether it is provided with two lists – which have ideally the same length – or a list and one value (see Figure 4.36). One list can be used twice (once shifted by an offset) as input for the makeSegment operation too (see Figure 4.37).

To formally describe this behavior, the *multiplicity* concept is introduced to the code graphs behind the GML Compositor. This concept is another interpretation of the definitions in the work of Thaller et al. [TKHF12, TKZ*13a] and is therefore equivalent to those definitions.

Definition 4.11 *The* ***multiplicity*** $m(t)$ *of a node type* $t \in \mathsf{NType}$ *is defined to be the dimensionality of a list type, or 0:*

$$m(t) := \begin{cases} m(t_0) + 1 & \text{if } t = \mathsf{List}[t_0] \\ 0 & \text{otherwise} \end{cases}$$

Definition 4.12 *The* ***multiplicity*** $m(n)$ *of a node* $n \in \mathscr{N}$ *is defined to be the multiplicity of its type* $\mathsf{nType}(n)$.

Definition 4.13 *The* ***excess multiplicity*** $\Delta m(e,i)$ *of the i-th source node of an edge* $e \in \mathscr{E}$ *is the difference between the multiplicity of the source node and the multiplicity of the corresponding input type of the edge label:*

$$\Delta m(e,i) := m(\mathsf{src}(e)_i) - m(\mathsf{edgeInType}(\mathsf{eLab}(e))_i)$$

Definition 4.14 *The* ***multiplicity*** $m(e)$ *of an edge* $e \in \mathscr{E}$ *is defined as the maximum excess multiplicity of all its source nodes:*

$$m(e) := \max_i \Delta m(e,i)$$

Informally speaking, an edge $e \in \mathscr{E}$ with positive multiplicity $m(e) > 0$ is equivalent to an edge $e' \in \mathscr{E}$ with multiplicity $m(e') = 0$, whose operation is wrapped into $m(e)$ nested FOR-loops.

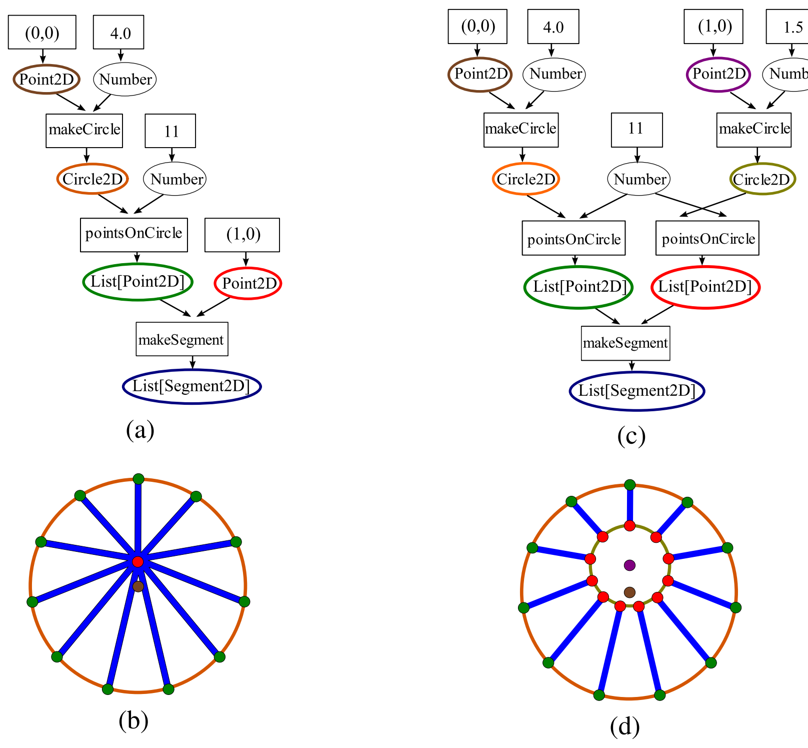


Fig. 4.36 Two simple procedural models ((b) and (d)) are shown together with their respective code graphs ((a) and (c)). Points, lines and circles in the models correspond to the equally colored nodes in the graphs, which represent intermediate results. As for the operations: makeCircle creates a circle out of a point and a radius, pointsOnCircle creates a list of evenly distributed points on a circle and a straight line segment between two points is created by makeSegment. Emphasis lies here on the makeSegment operation that generates the blue segments. This operation reacts accordingly to its inputs (a list of points and a point in (a) and two lists of points in (c)) and produces the right result. This adaption is done automatically by the system. Note that multiple graphical elements are represented by single code graph nodes. (image source: Thaller et al. [TKHF12])

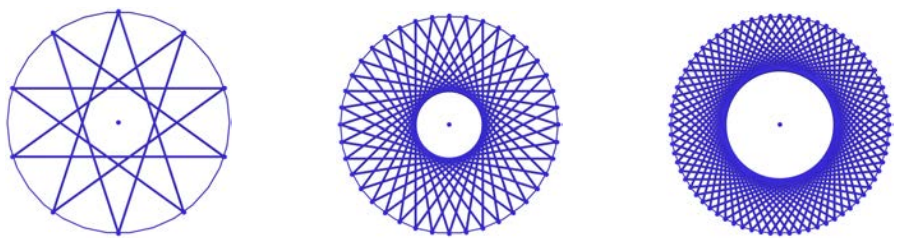


Fig. 4.37 The makeSegment operation applied to an amount of points distributed evenly on a circle (from left to right: 10 points, 40 points, and 72 points). The list of points – once shifted by an offset – is used for both inputs of the operation.

There are a few list-related operations that are convenient to use with the multiplicity mechanism:

Link $t\ t \rightarrow \mathsf{List}[t],\ t\ t\ t \rightarrow \mathsf{List}[t], \ldots$
The link operation collects all its inputs to create a list. All inputs have to be from the same type. All objects that are gathered by the link operation are treated in the same way from now on.

Specialize $\mathsf{List}[t]\ \mathsf{Number} \rightarrow t\ \mathsf{List}[t]$
The specialize operation defines an exception within a repetition. A list and an index is used to return, on the one hand, the element denoted by the index and, on the other hand, the remaining objects from the list as a new list. The single item can be processed in a different way afterwards.
Other specialize operations can be provided with advanced selection mechanisms, like selecting by ray casting or by smallest distance to a given point. In contrast to the index-based method, these alternative methods do offer a solution to the persistent naming problem.

Finally, there arise two major advantages due to the multiplicity concept. The effort necessary for the construction of objects that exhibit regularities and symmetries is greatly reduced. Nothing has to be done twice. This, furthermore, allows for easy re-parametrization. Parameters can be changed in an early construction step and changes are applied to all repeated parts.

Handling of Failures and Errors in the GML Compositor

Modeling operations do not always succeed. This is especially true in a procedural modeling framework; changing one parameter may influence all dependent operations and can violate, for example, size constraints. Subsequent operations can also depend on the amount of outputs, like when two circles are intersected with each other, two, one, or no intersection point can arise and dependent of the expected outcome further operations may become superfluous.

Such failures may have only a local effect and the whole execution does not need to be aborted because of a failure. When a target node is not filled by its corresponding edge, it carries an error value, which is propagated through the code graph. Operations that have error values in their source nodes are not executed. Not influenced operations are executed in the normal way. This behavior leads in most cases to the expected result; for example: When no intersection point is available, all constructions based on this points are not continued, or when there is not enough space to insert a special rule in a split grammar, the rule and all rules that follow are not applied.

Side Effects of Operations in the GML Compositor

Evaluating an mathematical expression with the same parameters usually denotes the same resulting value independent of when it is evaluated. However, in most commonly used programming languages expressions can have *side effects* that influence the result every time it is evaluated. A side effect could for example be a change to a global variable that is included in the calculations, and is therefore influencing the resulting value. Such side effects may introduce ordering constraints between operations within the code graph. If a global variable is changed and accessed by multiple operations, the order of execution becomes important to predict the outcome. This dependency, however, is not explicitly represented in the code graph, therefore side effects should be disallowed.

Consider, on the other hand, two different kinds of operations that are expected to be within a modeling framework: *Creation* and *refinement operations*. Creation operations add new objects to the scene and refinement operations modify objects in the scene. Such operations can be seen as to have side effects. They change the scene, which can be interpreted as a global variable. The two underlying side effects would be: adding an object to the scene and removing an object from the scene. A refinement operation can be seen as to remove the old object and replace it with one or more new objects. Objects are never really removed, as the scene is an accumulation of visible objects, removing an object just means to render it invisible, so that the information about this object still remains accessible.

There are ways to avoid these side effects. On the one hand, all visible objects can be returned as output of the procedural model. This, however, is not very intuitive as it clutters up the code graph with a large number of outputs and introduces a gathering operation that is connected to all of these. Even though the code graph is usually not visible to the user this will only lead to confusion should it be examined once. Houdini [Sid13] and Grasshopper [Rob13a], on the other hand, do not use any kind of side effects. Users have to explicitly mark nodes, which represent model parts, in the data flow graph to be visible in the final model. The GML Compositor avoids interacting with the underlying graph, so this solution is no alternative.

Incremental Updates of the Code Graph in the GML Compositor

When procedural models are manipulated in an interactive editor, usually, it is only a single parameter that is being modified. This parameter often only affects a small part of the whole model, so, for reasons of performance, re-evaluation of the entire model should not be necessary. Therefore, it is desirable to only perform the minimum amount of work that is required, i.e. only re-evaluate the operations that really depend on the input parameter that has changed. The straightforward way is to just evaluate all hyperedges that are consuming the changed value (directly or indirectly), which amounts to undoing all side effects caused by these operations, before re-executing them with the updated parameter. This method, however, is not sufficient enough in terms of performance. Our work [TKZ*13a] focuses on improving this concept in combination with the introduction of scene graphs (see Section 4.4.4) to the GML Compositor system.

The Problem of Aggregate Values. Re-evaluation can be very excessive whenever an input of a hyperedge in the code graph is of an aggregate type, like objects that consist of several independent parts. A list, for example, consists of a set of independent elements, so when a change in an input parameter occurs, it is not desirable to undo and recalculate all operations that use the unchanged elements. Further additional aggregate data types, which can also trigger excessive recalculations, can be added through new modeling vocabularies. Changing the material of an object should only affect the color of the object, but not its shape; so recalculating the unchanged geometry through re-evaluation is a lot more expensive than just changing the material.

The Realization of Scene Graphs in the GML Compositor. In particular, this problem affects the use of scene graphs (as described in 4.4.4) in the GML Compositor. The main strength of scene graphs is that it can be efficiently animated by changing the transformation of a node. This change then affects the transformation of all nodes in the affected sub-graph, without requiring a change of a part of that sub-graph. The code graph representation, however, incorporates dependencies that do not actually exist. Every scene graph node represented in the code graph is depends on its transformation, and all children nodes depend on their parent. Thus, a naive evaluation method would reconstruct the whole scene graph and reloads meshes associated to scene graph nodes every time when the transformation of the parent node is changed.

Incremental Updates for Individual Operations and Code Graphs. For a partial re-evaluation of a code graph the side effects of affected operations need to be undone before the operation can be executed with new parameters yielding again side effects. So for each modeling operation *op*, at least two further functions are required: $\mathsf{evaluate}_{op}$ and undo_{op}. The former executes the operation, which potentially causes side effects and the latter reverses those side effects.

$$\begin{aligned}\mathsf{evaluate}_{op}\ \ &(in_1 \cdots in_n) \to (\mathsf{S}_{op}, out_1 \cdots out_m)\\ \mathsf{undo}_{op}\ \ &(\mathsf{S}_{op}) \to ()\end{aligned}$$

Additionally to the output values of the modeling operation, the $\mathsf{evaluate}_{op}$ function also returns a state value S_{op} that serves as information for the undo_{op} function. For operations without side effects, the undo_{op} function does nothing and the state value S_{op} contains no information.

By taking advantage of previous calculations, re-evaluations can be optimized to allow partial updates of aggregate values. To incorporate this feature, modeling operation libraries may also provide an additional update_{op} function. Input and output values of the update_{op} functions are all annotated with a tag, used to identify whether a value has changed or not. Based on these tags, the update_{op} function then decides for what outputs a re-evaluation is necessary. For many data types it is sufficient to just track if the value has changed, but for scene graph nodes, for example, it is necessary to distinguish between different kinds of changes. A change that only affects the transformation has other consequences than other changes, which may trigger a complete re-evaluation of the subsequent operations.

To achieve this, a parametric data type $\mathsf{Tag}[t]$ is defined for each data type t. The different values for the $\mathsf{Tag}[t]$ data type are defined in correspondence to the individual requirements of the data type t. Every Tag data type has at least two special values NEW and $\mathsf{UNCHANGED}$ to support a default behavior. This update_{op} function is additionally supplied with the outputs of the previous evaluation of the operation. This is done to provide the tags for the newly calculated outputs of the corresponding modeling operation. This leads to following signature of the update_{op} function:

$$\begin{aligned}\mathsf{update}_{op}\ \ &(\mathsf{S}_{op}, in_1 \cdots in_n, \mathtt{Tag}[in_1] \cdots \mathtt{Tag}[in_n], oldout_1, \cdots oldout_m) \to\\ &(\mathsf{S}_{op}, out_1 \cdots out_m, \mathtt{Tag}[out_1] \cdots \mathtt{Tag}[out_m])\end{aligned}$$

If the update_{op} function is not defined for a modeling operation, a default implementation based on the undo_{op} and $\mathsf{evaluate}_{op}$ functions is provided, which re-evaluates the whole operation if one tag carries the NEW value, or skips re-evaluation if all tags carry the $\mathsf{UNCHANGED}$ value.

For re-evaluation of a code graph there exist the operations $\mathsf{evaluate}_G$, undo_G, and update_G. These operations call the corresponding functions for each edge in an arbitrary topologically sorted order. The individual operations are passed the state values that are returned from the corresponding functions. A complete traversal for updating a code graph can be avoided based on the assumption that the individual update functions will return UNCHANGED values when all their inputs are UNCHANGED.

4.4.3 Interactive Split Grammar Modeling

The GML Compositor was inspired by the domain of split grammars, especially ones based on convex polyhedra (see Section 4.2). Even though grammars are a different formal language, the underlying grammar rule set can be mapped to code graphs. Rules with a predefined number of inputs and outputs directly map to code graph operations. In other cases, where the number of outputs depends of the size of the input shape the results are collected in an array and further processed as repeated elements.

Modeling within the GML Compositor usually is done in a top-down manner. It starts off with a coarse approximation, which is subsequently refined to obtain the desired result. In the course of the realization of split grammars in the GML Compositor, the volumetric entities reminiscent of non-terminal shapes carry a local coordinate system and are composed of convex polyhedra. In this section these volumetric entities are referred to as *shapes*. This modeling vocabulary has been introduced by Thaller and Krispel. I extended the library of modeling operations and procedural assets in collaboration with them.

In classical architecture repetitions and symmetry is common. Thus, it is often necessary to apply the exact same sequence of operations to several objects. The multiplicity mechanism (see Section 4.4.2) of the GML Compositor gathers different shapes into an array to treat those shapes the same and reduce the workload. This array of shapes will be referred to as *link group*. Operations applied to an element of a link group are automatically applied to all other shapes in the group as well. These operations are all applied using the shape's respective local coordinate system. All preceding modifying operations (e.g. mirroring operations) on a single shape of the link group still affect only that single shape. The combination of mirroring one shape and linking it to another can be used to model symmetry. The main difference here, in comparison to conventional systems, is that geometry is not simply copied and scaled, but evaluated in the current shape's state instead.

Interactive Split Grammar Modeling Operations

The GML Compositor provides a set of twelve basic modeling operations. These operations are divided into four groups: *creation operations* (generate new shapes), *refinement operations* (create sub-shapes), *grouping operations* (treat shapes equally) and *modification operations* (change shape's state).

In the remainder of this part these twelve modeling operations are explained in detail (inspired by our work in [ZKT*14]). Most operations are accompanied by a schematic, abstract illustration as well as a more elaborated example.

Interactive Creation Operations. The creation group contains operations for shape creation and removal:

Box.
"Create a box shape". A box shape (of adjustable dimensions) is created as a starting point for the modeling process.

Void.
"Make a shape invisible". The opposite of the box operation. The selected shape is rendered invisible and cannot be refined further (but it still exists internally).

Interactive Refinement Operations. The refinement group provides operations – some of which are inspired by standard split grammar operations (see Section 2.3.4) – to partition a shape in smaller shapes and to add detail.

Subdivide.
"Split a shape into parts with a given list of sizes". This operation is equivalent to the subdivide operation used in split grammars. The selected shape is split along a given direction into smaller parts. Planes orthogonal to the given direction are used for splitting. The sizes of the sub-parts are defined by a set of sizes along the split direction. These can be defined as combinations of relative proportions and absolute measurements. Figure 4.38 illustrates the usage of this operation.

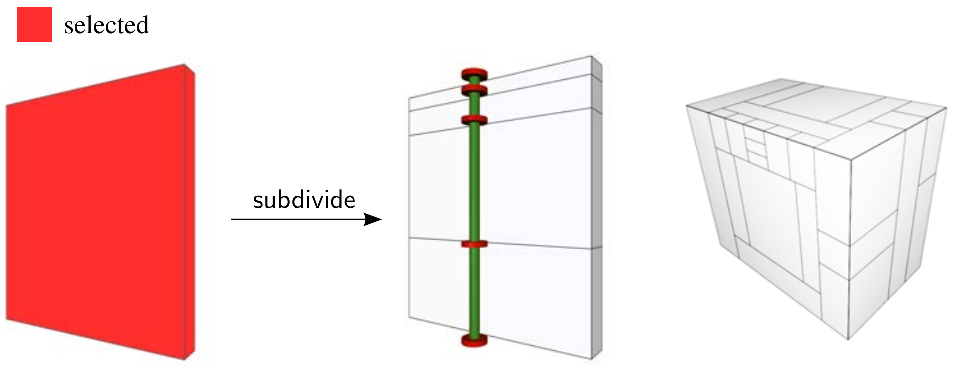


Fig. 4.38 **Subdivide**. A shape is split into four parts along a given direction through applying the subdivide operation. The size of each part can be defined using relative proportions or absolute measurements. The (red) discs can be dragged to change the split intervals interactively. This operation can be used to create arbitrary hierarchical decompositions in three dimensions as shown on the right. (image adapted from Zmugg et al. [ZKT*14])

Extrude.
"Displace faces of a shape in a given direction". The faces of the selected shape with a normal similar to the given direction are displaced by a certain given amount. A new shape is created for the extruded part. Figure 4.39 shows the application of the extrude operation.

Diagonal Split.
"Split a shape along one of the diagonals of the shape's bounding box". Introduces a diagonal split to the selected shape and divides it in two parts. There are six possible cases; the selected case is specified as a parameter.

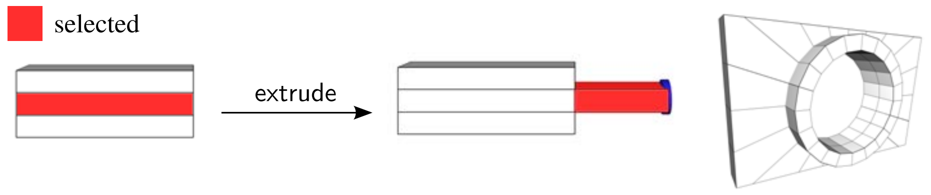


Fig. 4.39 **Extrude**. The selected shape is extruded in the given direction producing a new shape. Dragging the (blue) discs modifies the extrusion offset interactively. This operation can be used to enable three-dimensional details like borders as shown on the right. (image adapted from Zmugg et al. [ZKT*14])

Repeat.

"Split into linked parts of given size". As within split grammars, the selected shape is split into the maximum number of parts along the provided split direction with the given minimal size. The resulting shapes form a link group (see Figure 4.40).

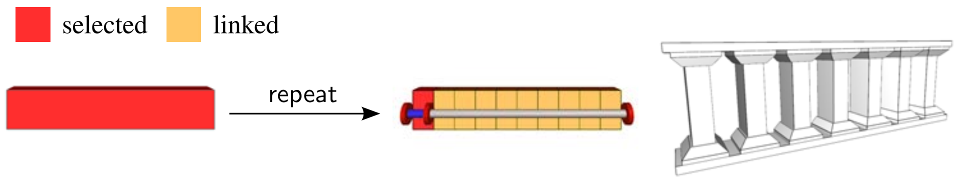


Fig. 4.40 **Repeat**. The repeat operation splits a shape into equally sized parts, given a minimum size and a direction. The space is distributed evenly among the elements. Changing the minimum size can result in a change of the number of elements. The resulting elements are all contained in the same link group; therefore modifying any of these will result in modifying all. Dragging the middle (red) disc modifies the minimum size of the result shapes. A sequence of pillars can be realized using this operation as seen on the right. (image adapted from Zmugg et al. [ZKT*14])

RepeatABA.

"Split into alternating sequence of parts A *and* B *of given sizes"*. The selected shape is split along the provided split direction into an alternating sequence of parts. The sizes for each individual part is determined by the given two sizes for the groups A and B. Either size of the parts of the group A or B can be set to a fixed size, any remaining space is distributed evenly among the elements of the opposite group. An additional parameter determines the latter group. The sequence begins and ends with an element of the group A. The results are two link groups; all A-parts are treated the same, and so are the B-parts. This can be used, for example, for an automatic distribution of pillars and gaps in a shape.

Merge.

"Merge shapes into one". Several selected shapes are joined together to form one new shape. This is the inverse to the refinement operations.

Interactive Grouping Operations. Thirdly, the grouping operations provide methods for processing several shapes at once and are based on the aforementioned operations in connection to the multiplicity mechanism.

Link.
"Treat these shapes in a similar way". A link group is created from the selected shapes. Any operation applied to one shape of the group is automatically applied to the other shapes – in their respective local coordinate system – as well. Figure 4.41 shows the behavior of linked shapes.

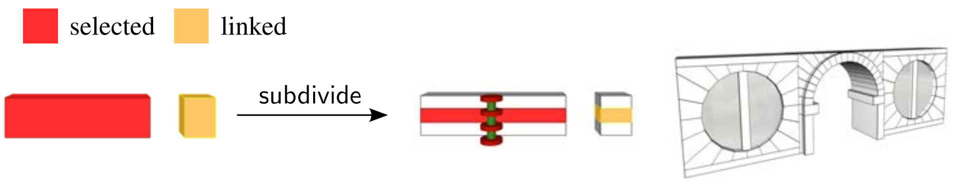


Fig. 4.41 **Link**. The link operation allows applying operations to groups of objects: Two shapes have been linked, and a subdivide operation is applied to the selected shape. The linked shape automatically mimics the same split; three pairs of linked shapes are generated from one linked pair. All constructions are repeated automatically on linked shapes as shown on the right; no redundant work is necessary. (image adapted from Zmugg et al. [ZKT*14])

Specialize.
"One (or more) shapes in a link group are exceptional". This is the inverse of the link operation. One or more shapes are released from the link group. Formally, the link group is partitioned into two disjoint link groups. Each group can then be refined in a different way.

It is important to emphasize the difference between the merge and link operation. Figure 4.42 illustrates this difference. Linked shapes behave similarly, so if a linked shape is split in half, so is each scope in the link group individually. The splitting plane may differ when the shapes have different sizes. In contrast, merged shapes have become one single shape and consequently there is only one splitting plane.

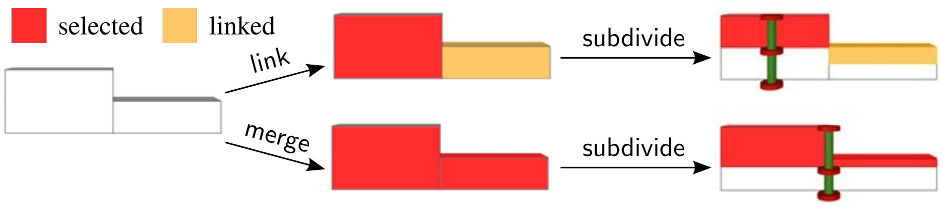


Fig. 4.42 Difference between link and merge operations: A subdivide operation is applied to a link group of shapes (top row) and to merged shapes (bottom row). In the link group, the split operation is applied to each shape individually. After merging, the two shapes have become one and consequently there is only one splitting plane. (image adapted from Zmugg et al. [ZKT*14])

Interactive Modifying Operations. Finally, modifying operations do not change the geometry of the shapes, but only its state, e.g. its local coordinate system. All subsequent operations on the shape are influenced by these changes.

Mirror.

"Mirror the orientation inside a shape". Modifies the local coordinate system, given a mirror direction (x, y or z-axis of the local coordinate system). Figure 4.43 shows the behavior of mirrored shapes.

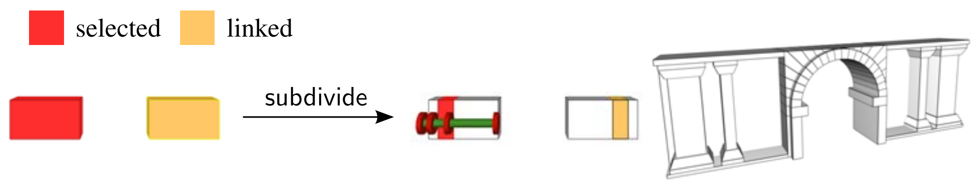


Fig. 4.43 **Mirror**. This figure illustrates the behavior of subdivide on mirrored and linked shapes. To achieve mirrored geometry one part of the link group is mirrored before linking. A subdivide operation is applied to the selected shape of the link group, resulting in three mirror linked shape pairs. The symmetry of the right and left parts of a simple model shown on the right was realized by mirroring and linking the coarse partitioning beforehand. (image adapted from Zmugg et al. [ZKT*14])

Rotate.

"Rotate the orientation inside a shape". This rotates the local coordinate system by 90 degrees given a rotation axis (counter clockwise or clockwise).

Procedural Assets for Interactive Modeling. In addition to the low-level modeling operations described above, the system includes a set of pre-modeled assets, i.e. a set of ready-made window parts (see Section 5.2.3), arches, and other architectural elements that can be inserted into a façade. They are, however, not the usual static three-dimensional models, but parametric models. Thus, they can approximate a wide range of actual architectural elements by adjusting various parameters and adapt automatically to different sizes and proportions when applied to shapes with different dimensions.

The distinction between low-level operations and high-level parametric assets is to some degree arbitrary; some asset operations can also be seen as more sophisticated split operations, as they partition a shape into parts that can be refined later on. One example is the round-hole operation as described before in Section 4.2.3. Such parametric partitioning assets can be very versatile because the inner and outer part can be further refined. The round-hole operation can be applied subsequently to produce detailed architectural elements such as windows with borders, or it can be used to produce a round pillar structure (see Figure 4.44 for reference).

Interactive Split Grammar Examples in the GML Compositor

A Simple Interactive Split Grammar Example. Figure 4.45 shows the result of a simple split grammar with the corresponding (simplified) code graph. This grammar describes a simple building whose amount of windows and stories adapt to the input shape. To focus on the

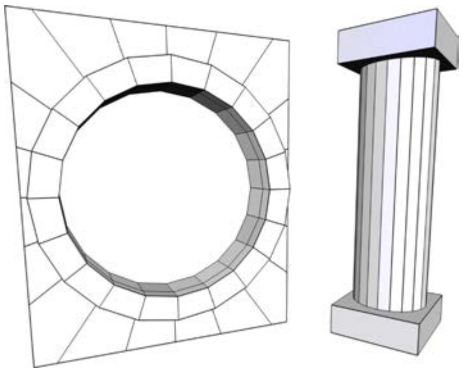


Fig. 4.44 The round-hole operation, as described before in the context of split grammars on convex polyhedra in Section 4.2.3, can be very versatile because the inner and outer part can be further refined. So it can be, for example, used sequentially to create a round window with borders, or it can be applied to create a round pillar structure. (image source: Zmugg et al. [ZKT*14])

mapping of shape grammars to code graphs, the door tile is split off before the repeat operation is applied to obtain the different stories. An alternate way of describing this model is to gather all walls into one array, perform the repeat split into the stories and then define an exception for the entrance using a specialize operation.

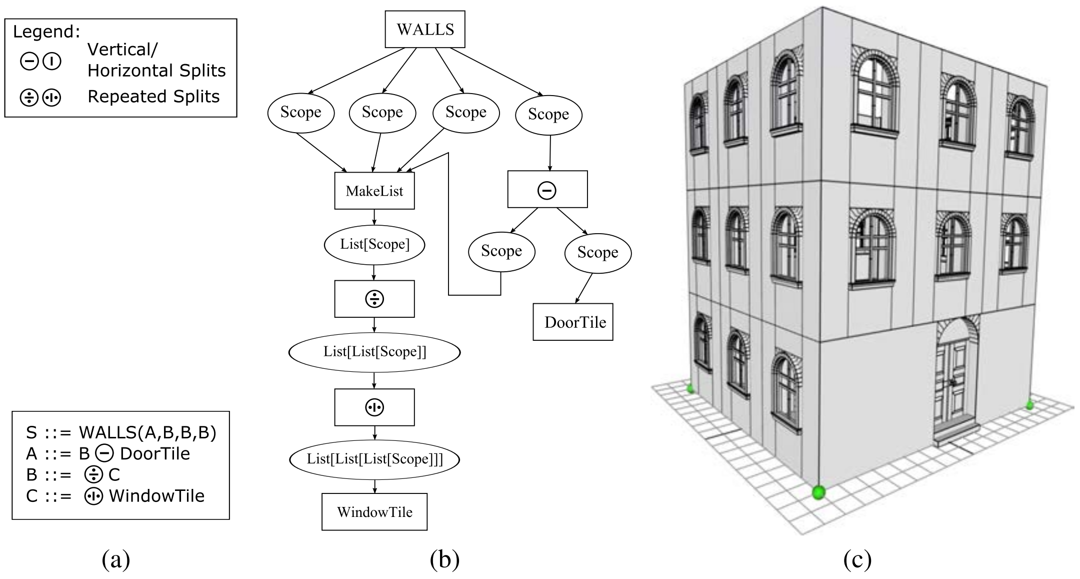


Fig. 4.45 Split grammars are usually defined by a textual description (a). The hierarchy that results from the application of the rules can be described by a code graph (b), whose execution leads to a three-dimensional model (c). Note that the code graph is simplified for brevity, thus no split rules for doors and windows are shown. (image source: Thaller et al. [TKHF12])

An Example for Interconnected Structures. One major drawback of (context-free) split grammars is the lack of mechanisms that connect structures across different parts of the top-down modeling hierarchy. The solution proposed by Krecklau and Kobbelt [KK11] uses nested arrays to which potential connecting points are appended as side effect. These points later processed to generate connecting geometry. This method can directly be mapped to the multiplicity framework, with the difference that connecting points have to be explicitly linked (using the link operation). Instead of receiving a container and explicitly adding objects to it, objects are linked explicitly and are passed as nested list to operations. Advantages here are that, in contrast to Krecklau and Kobbelt, there is always a visual representation in the system. Furthermore, passing the same container to all operations enforces a linear execution order, which conflicts with the idea of direct manipulation interfaces as mentioned before. An example of interconnected structures achieved in the GML Compositor is shown in Figure 4.46.

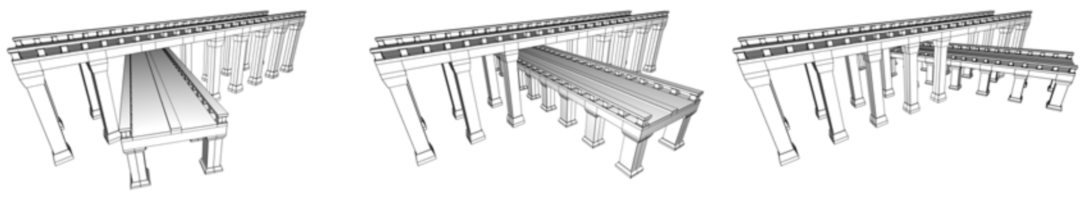


Fig. 4.46 The pillars of the bridges are constructed using ray casting for obstacle detection. Three different positions of the lower bridge showcase the interactive specialization within the repetition of bridge pillars. (image adapted from Thaller et al. [TKZ*13a])

4.4.4 Interactive Scene Graph Modeling

Three-dimensional scenes with inherent hierarchical structures, which means that individual objects are placed in relation to a parent object, are usually described by a scene graph (see Section 3.6). Figure 4.47 shows a scene where pieces of furniture are placed in relation to the room in which they are located, and the objects on and around a table are placed relatively to the table. Each scene graph node contains a transformation and, optionally, geometry. By varying these transformations over time, animations based on the scene graph can be achieved quite easily. The transformation applied to each piece of geometry is the product of all transformations on the path from the root to the specific node. Information of objects that occur more than once, like the chairs in the current example, are only stored once in the scene graph, which makes the scene graph an acyclic graph instead of a tree. The extension of scene graphs to the GML Compositor is mainly contributed by me and has been discussed in our journal paper [TKZ*13a].

Interactive Scene Graph Modeling Operations

To include a scene graph in the GML Compositor, a type Node is needed to represent various kinds of scene graph nodes. It is assumed that one instance of this type – the *root* node – is passed to the code graph as input and is always present in the scene. Child nodes are created by operations that take a parent node as input. In particular, there is a createNode operation that creates a node with only transformation information and no visible geometry, and a loadGeometry operation that creates a node with geometry that has been loaded from a file. Finally, to reference absolute positions in scene graphs, either to set up cameras or to connect objects that reside in different parts of the scene graph, a toGlobal operation exists, which converts a point

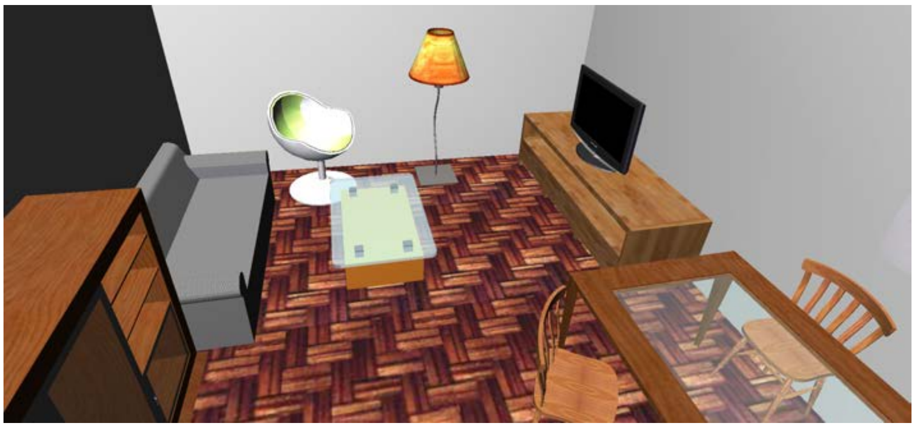


Fig. 4.47 Scene graphs allow the representation of hierarchical dependencies; as the TV is placed in dependence of the table, changing the table's position will also move the TV accordingly. Furthermore, scene graphs are memory efficient through instancing: the two chairs in this scene refer to the same geometry with different transformations. (image source: Thaller et al. [TKZ*13a])

from the local coordinate system of a scene graph node to global coordinates. By combining the createNode and toGlobal a createNodeAt operation that places a child node at a point given in global coordinates can be provided. Real world applications may need further node-creating operations, but the basic structure will remain the same.

For animated scenes, there are also basic operations to define key frame animations, one of which is called interpolatePose. Hereby a specific scene graph node's transformation is interpolated between the two transformations of two other nodes. This all happens in a specific time frame which is provided additionally. To support different camera views and animated camera flights, a camera object can also be placed on scene graph nodes, which again can be animated. Camera objects store a view direction; position and orientation are defined by the scene graph. With a designated interpolateCamera operation is is also possible to just interpolate between two distinct camera views.

Figure 4.48 shows a simple scene in the context of virtual museums. This scene is inspired by our work [ZTH*12], in which we facilitate the use of scene graphs in the GML Compositor. This figure shows the corresponding code graph for the scene on the right side. The inputs for this code graph are the number of museum exhibits to be shown and a list of paths to files containing the three-dimensional models of the exhibits. The requested number of models is loaded from the list and placed in correspondence to the scene graph nodes, which are arranged in a circle.

Interactive Modeling with Scene Graphs. For interactive modeling of a scene graph, a set of widgets is provided by the GML Compositor. These widgets allow the user interface to present standard scene graph semantics to the user. Our work [ZTH*12] describes this from an application's point of view.

Scene graph nodes are represented graphically as a widget (Figure 4.49(a)) that consists of three parts: the cylinder part for moving parallel to the ground, the arrow part for lifting, and the ball part for rotation. During motion arrows indicate the hierarchy, i.e., the parent node and the (direct) child nodes (Figure 4.49(e)). To make stacking easier, when lifting a node, it snaps to the parent level, and to the uppermost plane of the bounding box of the object positioned at the parent node. To have more control over rotations along all three main axis a separate

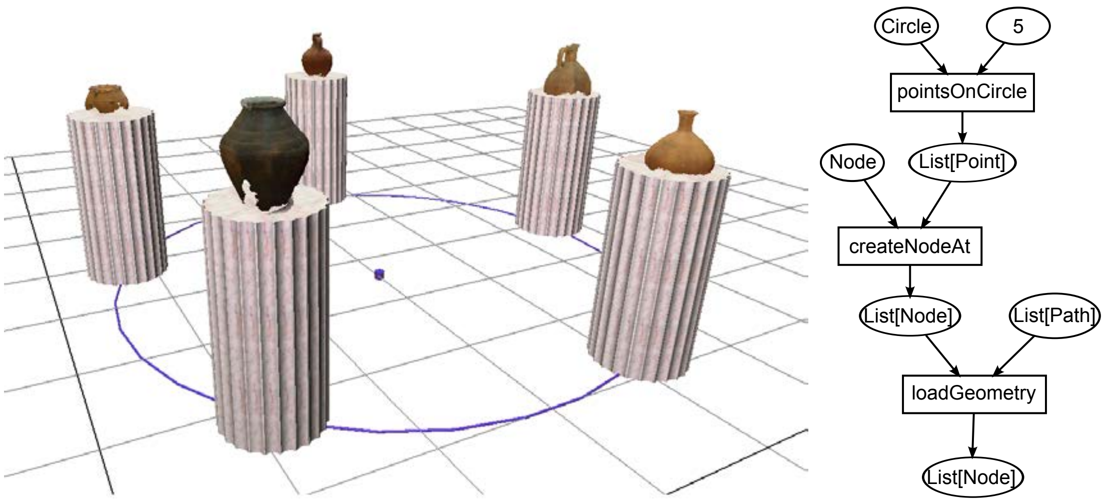


Fig. 4.48 A procedural scene graph: Scene graph nodes with pedestals are placed at points distributed on a circle (left). On top of each pedestal, a scanned model is placed. The inputs for the code graph (right) that represents this procedural model are the number of models and a list of file paths to load the models from. For simplicity, the operations representing the pedestals have been left out in the code graph. (image source: Thaller et al. [TKZ*13a])

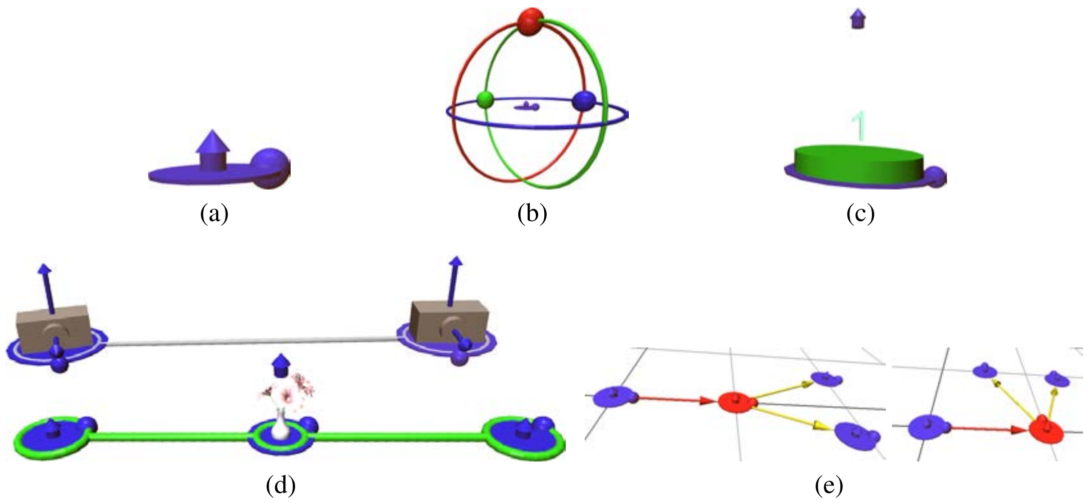


Fig. 4.49 The five widgets used for scene manipulation: scene graph node (a), full rotation widget (b), drop target (c), motion path widgets for interpolations (animation of objects and camera) (d) scene graph hierarchy (e). Understanding these five concepts (plus time) is sufficient for constricting animated scenes in the GML Compositor. (image source: Zmugg et al. [ZTH*12])

rotation widget is provided (Figure 4.49(b)). A place-holder for not yet available objects is provided through a special widget called *"drop target"* (Figure 4.49(c)). Once the necessary objects become available, these widgets can be filled automatically. In our work[ZTH*12] they have been used to define templates for animated museum exhibits. More information on this is available in Section 6.4. Finally, to visualize animations that have been defined in the scene, motion path widgets (Figure 4.49(d)) illustrate the path along the objects move.

Synopsis

In this chapter I presented procedural techniques that were either utilized or developed in context of this thesis. This includes – first and foremost – the scripting language GML. This Adobe PostScript dialect has been designed for describing three-dimensional shapes. The operator sets available in this scripting language can be extended to include arbitrary shape domains. I used and extended the GML to create all results presented in this thesis.

The core part of this chapter contains the description of two extensions to the state of the art of split grammars. The first extension describes the generalization of standard box-based split grammars to split grammars based on convex polyhedra. Through this, a more general class of complex shapes is made amenable to the grammar formalism without the need to include pre-modeled geometry. The second extension – my main contribution – describes how curved and deformed architecture can be described efficiently through split grammars. Through the inclusion of free-form deformations to the grammar formalism, the amount of results of replacement rules can adapt to the space provided through different deformations. I, furthermore, discussed how deformed shapes can be further processed to, for example, split deformed shapes in correspondence to the deformation or by arbitrary straight planes.

The last section in this chapter discussed an interactive procedural modeling tool, the GML Compositor. This tool has been designed to create procedural models without the overhead of programming. It supports different modeling vocabularies, among which are a split grammar inspired tool set and a scene graph tool set. The latter tool set has been included by me to enable the creation of animated procedural environments, which have been featured in our projects.

Chapter 5
Case Study: How to Formalize a New Shape Domain

Contents

Abstract. Before one can start the task of generating a library of procedural models to cover a certain domain, it is necessary to analyze the chosen domain. As part of the analysis of one of the most imposing buildings in Paris, the Louvre, a case study on the domain of windows is performed in this chapter. Windows are not chosen arbitrarily for this case study; they are among the most salient features of façades and are a combination of different inter-related design elements, which makes them a prime example for procedural modeling. This chapter first details the progress in the analysis of the bigger reconstruction task: the Louvre in Paris, and ultimately concludes in a detail analysis of windows. This analysis eventually leads to the formulation of the Generative Fact Labeling method. Through this method a way is demonstrated how to formalize a shape domain and generate a library of procedural building blocks.

5.1 Reconstruction of the Louvre

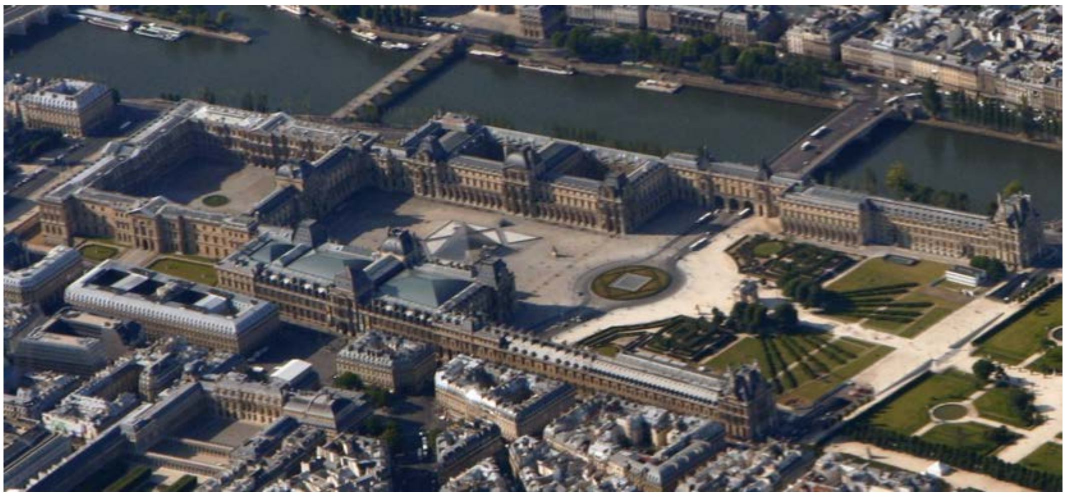


Fig. 5.1 Aerial photograph of the Louvre Palace in Paris. (photograph copyright by Matthias Kabel, license: CC BY-SA 3.0)

The Louvre in Paris as shown in Figure 5.1 is a prime example of classical architecture. It is of non-trivial size and complexity and is a monumental historical building. To evaluate the scalability, as well as the applicability in the domain of Cultural Heritage, of the GML Compositor (see Section 4.4), we attempted a partial reconstruction of this building in our work [ZKT*14]. The whole façade reconstructions were mainly done in context of the Bachelor's thesis of Pszeida [Psz14] in order to estimate the usability and effectiveness of the GML Compositor software. Illustrative step-by-step examples have been done by Krispel and me to showcase the individual tasks that are necessary.

This reconstruction is not the first attempt at a procedural description of parts of the Louvre. With Esri's CityEngine [Esr13] Calogero and Arnold [CA11] generated two alternative proposals of Perrault's Colonnade in east wing of the Louvre using procedural modeling. Their analysis of the different split layouts in the Le Vau/Perrault/Le Brun design of 1668 of the east wing as it stands today, served as basis for the reconstruction done with the GML Compositor. The benefit for the Cultural Heritage community is that the GML Compositor scales better than conventional forward modelers like Google SketchUp [Goo13c], but does not require actual coding like Esri's CityEngine [Esr13].

The following sections give an overview of the available source material and use existing architectural analyses to reconstruct a specific façade of the Louvre step by step as presented our work [ZKT*14]. The final reconstruction results are presented in the upcoming chapter in Section 6.2.1.

5.1.1 Source Material for the Louvre Reconstruction

For a reconstruction of a building it is first necessary to acquire sufficiently accurate source material. Not accurate enough measurements lead to significant problems with conventional three-dimensional modeling software packages, as inaccurate measurements can be corrected only with excessive time and effort. However, with a parametric model, all parameters can be changed to the correct values at any time they become known. In case of the Louvre reconstruction, some of the original building plans, which provide ground layouts as well as façade layouts for parts of the Louvre had been accessible. In total, our group had access to eight ground plans and nine façade plans of different parts of the Louvre. They cover parts of the Cour Napoléon and the Cour Carrée (see Figure 5.2). Although much detail is provided in these plans, they lack the extrusion depths of the façade elements. These depths were therefore estimated from photographs. Such incomplete information is typical in such reconstruction tasks and emphasizes the importance of the procedural approach.

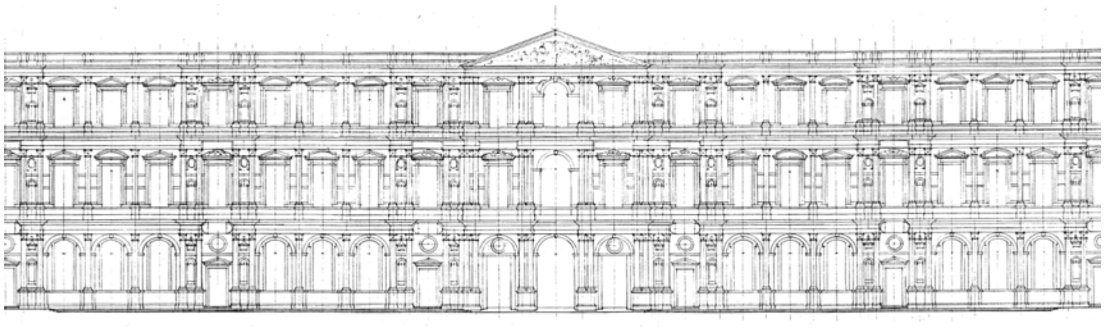


Fig. 5.2 Detailed façade layout of one of the façades of the Cour Carrée. (image source: Archives of the Louvre)

5.1.2 Stepwise Reconstruction of a Louvre Façade

This part describes the methodology for creating a procedural reconstruction of the pavilion part of Perrault's Colonnade on the east side of the Louvre (see Figure 5.3) in the GML Compositor. This model is based on the split proposal given by Calogero and Arnold in their work [CA11]. The parametric assets for the windows have been acquired through the analysis done in the upcoming Section 5.2.

Constructing a Rough Layout.
First, a rectangular block representing the pavilion is partitioned into the vertical structure of *podium*, *pilastrade*, *entablature* and *parapet* (see Figure 5.4(a)) using the interval-split operation. Next, the parts that are not part of the entablature or ledges are split into three horizontal parts, which reflect the coarse horizontal structure of the façade (see Figure 5.4(b)). Furthermore, the parts on the right side are mirrored and linked to the left parts to reflect the symmetry of the façade. Consequently, only one side has to be modeled to create the symmetric parts. More detail is added to the façade parts by splitting down to the hierarchy level of windows and pillars (see Figure 5.4(c)). The correlation of the partitions of pilastrade and parapet was realized with splits that use the same parameters as input.

Reconstruction of the Podium.
The podium part is refined first. The lower podium part is split off and extruded. The upper part is split into window and wall parts. The window parts are linked, due to the similarity of the three podium windows. Window detail is added by applying different parametric win-

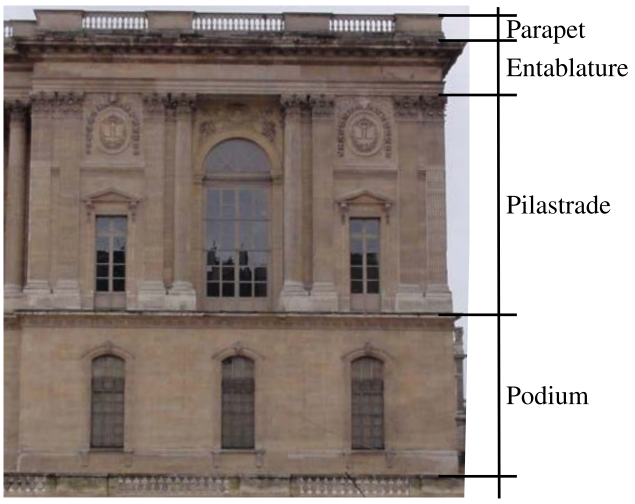


Fig. 5.3 Split analysis of Calogero and Arnold [CA11] illustrated in a photograph of the pavilion part of Perrault's Colonnade in the Le Vau/Perrault/Le Brun Design of 1668. (photograph courtesy of Erica Calogero)

dow assets. These include assets for the window top with an inscribed key stone, the window frame, the window sill, as well as the crossbars (see Figure 5.4(d)).

Reconstruction of the Ledge.

The ledge is modeled using a series of extrusions. First, the vertical partition is split and extruded to the front and sideways. The corner parts are modeled by again extruding the sideway parts to the front. The slanted parts are created using diagonal splits; corner parts are split twice using diagonal split (see Figure 5.5(a)). The intermediate result can be seen in Figure 5.4(e).

Reconstruction of the Pilastrade.

For the pilastrade part, the pillars are first split vertically into base, column and capital parts. The base detail is created using diagonal splits, and the fine alternating structure of the rectangular pillars is created using the repeatABA operation. The windows on the side parts are created using parametric assets for the border and the top parts (see Figure 5.4(f)). The ledge above the window is created by linking the space between the pillars and above the window and applying extrusions and splits to one of the parts of that link group.

The middle part is first split into depth direction, and the front part is voided to create the depth offset. The big window is again modeled using parametric assets. Additional details are added using splits and extrusions. The pillars need special handling due to different wall depth on the left and right side of the pillar. The correct depth is acquired by splitting the space around the pillar into four parts, setting the according depth and voiding the outer parts as illustrated in Figure 5.5(b). The intermediate result can be seen in Figure 5.4(g). The capital part of the pillars is roughly approximated using parametric assets.

Reconstruction of the Entablature and the Parapet.

Finally, the entablature is modeled similarly to the ledge above the podium, just with more extrusions. The solid parts of the parapet are linked, and their top part is extruded to create the ledge; the pillar rows are created using the repeat operation. The final result is shown in Figure 5.4(h).

This concludes the detailed description of the reconstruction of the pavilion part of Perrault's Colonnade. The other façades can be constructed in a similar way in a coarse to fine manner. The most distinguishing features are the windows featured in these façades. The upcoming section focuses on the generation of a library of procedural window building blocks that have been used in the reconstruction of the Louvre façades.

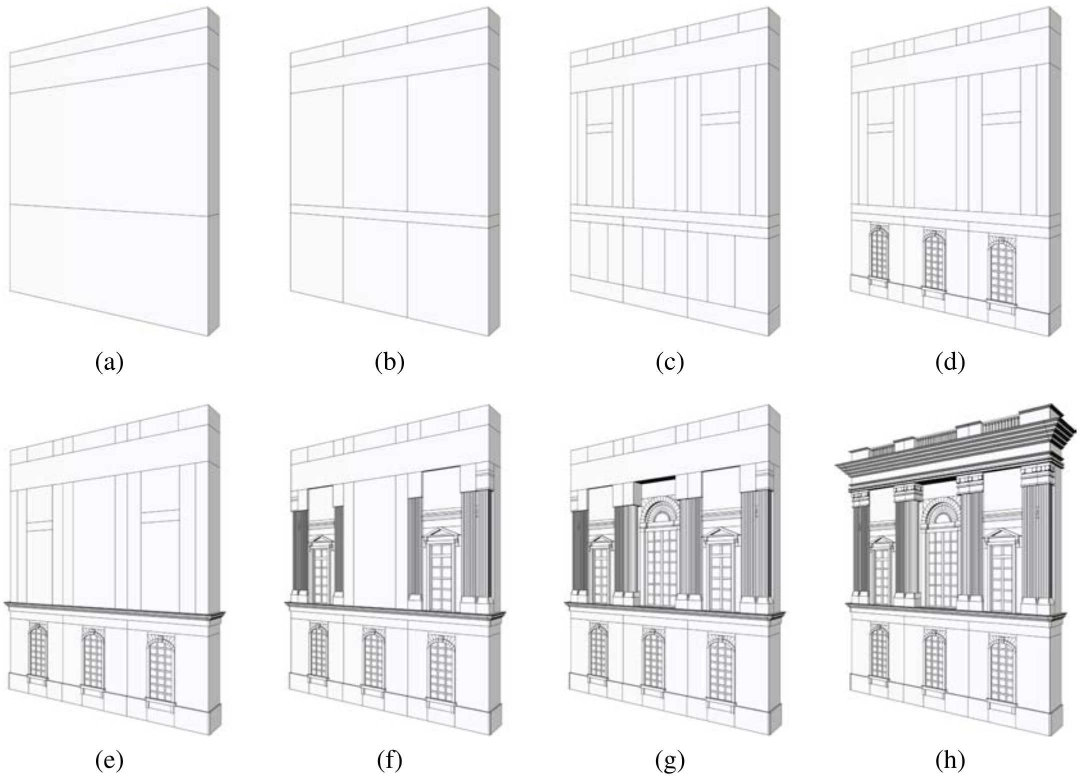
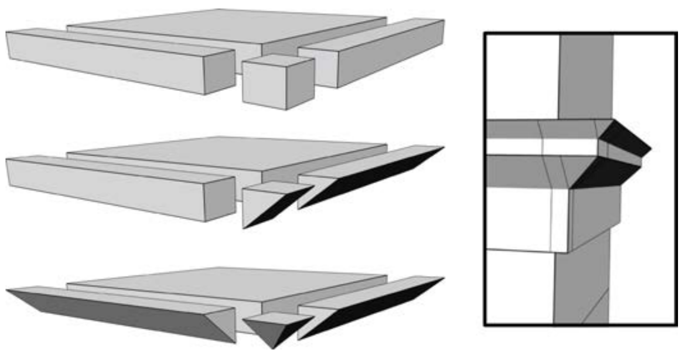


Fig. 5.4 Intermediate steps of the reconstruction of the pavilion part of Perrault's Colonnade on the east side of the Louvre. The steps range from coarse vertical and horizontal split structure ((a) - (c)), over work on the podium (d), ledges (e), and pilastrade ((f) - (g)) up to the final result (h). (image source: Zmugg et al. [ZKT*14])

5.2 A Library for Procedural Window Building Blocks

Windows are among the most salient features of façades. They are more than just a rectangular hole in a wall; in classical styles of architecture a window is a combination of different interrelated design elements, which may derive from a long-standing architectural tradition. This complexity leads to a significant amount of time needed for the three-dimensional reconstruction. Therefore, most digital urban reconstructions today suffer from bad window representations. These windows are either well modeled, but do not match the originals due to insufficient accuracy with the use of fixed pre-modeled assets libraries, or are simply not modeled at all and are replaced through the use of a photo-texturing.

Our work [TZK*13] focuses on these issues and offers an improvement for the quality of reconstructed windows. This section is mainly based on this publication, in which a method-


(a)

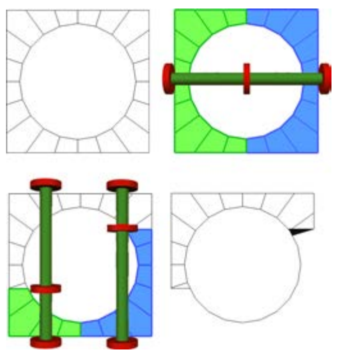

(b)

Fig. 5.5 Ledges (a) are modeled using extrusion, diagonal splits and voiding the unnecessary parts. Note that the corner part is split twice using diagonal split (left column, top to bottom). This method can be used to model more complex profiles as shown on the right. For modeling a pillar (b), which is adjoined to walls with different depth offsets, first, a round-hole operation is applied (upper left). Then, the outer part is split into two parts (upper right), to which the different depth offsets are applied using another split operation (lower left). Finally, the outer parts are voided (lower right). (image source: Zmugg et al. [ZKT*14])

ological approach to capture a large number of highly structured, similar but not identical shapes, is presented. This so-called *Generative Fact Labeling* (GFL) method is composed of three stages:

Analysis Phase:
To acquire a broad basis of different windows, it is necessary to concentrate not only on the windows of the Louvre. Thus, a collection of 150 photographs of complex windows in the city of Graz, Austria, has been gathered. These window exemplars have been structured by assigning fact labels.

Synthesis Phase:
Through analysis of these windows, a library of combinable procedural assets has been created. These procedural building blocks correspond to the elements identified in the analysis phase.

Verification Phase:
In order to assess the usefulness of the procedural library several well-chosen windows from this collection have been reconstructed and compared to the original exemplars.

The goal of this method is not only to produce a library of functions that allow reproducing the limited number of given exemplars, but also to formalize the design space that is spanned by them. Is is important to be aware that this factorization is only one possible interpretation. The presented method is more of a guideline how to find – at least – a reasonable procedural explanation of a complex shape class.

In terms of windows, Chevrier et al. [CCGP10] did related research. They investigated how practical parametric components in Cultural Heritage reconstruction tasks are. They created a Autodesk Maya [Aut13c] plugin, where components are instantiated using a first estimate by a user. This initial estimate is further refined with the help of photographs, plans, and point clouds. In comparison to our work, they focus more on the modeling process, while our method focuses more on the analysis aspect.

The Generative Fact Labeling method emerged from discussions within our research group on the Bachelor's thesis of Posch [Pos13] on the topic of procedural window building blocks.

The following sections focus first on a definition of the Generative Fact Labeling method and then apply the method's three steps to the domain of windows in separate sections. Further results and comparisons are discussed in the upcoming Section 5.2.4.

5.2.1 The Generative Fact Labeling Method

The Generative Fact Labeling method (see Figure 5.6) is composed of three stages that are explained in this section in detail. In the following sections this method is then applied to the domain of windows, which eventually results in a comprehensive library of procedural window building blocks.

The Analysis Phase. The starting point of the process of generative shape reconstruction is, in general, a finite collection of exemplars. An exemplar, which is an indisputable *fact*, is associated with a set of *observations*, which are human interpretations of these facts. To structure the observations, related observations of different exemplars are grouped together. A basic observation, for example for a window, is that it consists basically of a hole in the wall. The shapes of these holes are mutually exclusive; a hole is either rectangular shaped or circular shaped, but not both. Mutually exclusive observations lead to a set of alternatives, which build a *label group* carrying a *label*. For simplicity reasons capital letters (A,B,C) are used for labels and enumerate all possible alternatives. The alternatives are then labeled A1, A2, A3, ... for a label group A. The ensemble of observations, labels and label groups is called a *classification* of the exemplars.

Unfortunately, it is not reasonable to assume that the set of labels in each group, nor the set of groups, will ever be exhaustive. By following the *open world assumption*, it is clear that the set of exemplars is not complete and may always grow. Consequently two special labels for each group are added. These are *not applicable* (A$-$), and *not yet expressed* (A$*$). The former denotes an observation that that corresponds to a label A is not present, and the latter indicates that something that corresponds to a label A is there, but none of the available alternatives apply.

The fact labeling method typically proceeds in a coarse to fine manner. First rough structures, such as layouts, are determined and fine structures explored afterwards to differentiate between alternatives more locally. The goal is to decouple the labels, but no hierarchy between the labels is assumed, but the relation between label groups can turn out to be quite complex.

The Synthesis Phase. After extracting label and label groups and identifying similarities between different label groups it is necessary to construct procedural assets based on them. In this step it is important that these assets are designed in a way that they can be combined and adapt to different size constraints appropriately. The instances of the corresponding variants that are present in the exemplars should server here as marginal cases of the procedural models. Transitions between different instances should always produce valid configurations.

The Verification Phase. The final step is then to combine the procedural building blocks to create instances of the analyzed objects. By reconstructing exemplars and comparing them to the originals, the usefulness of the procedural building blocks can be assessed. The whole process is in no way linear: if it becomes clear in the verification phase that a building block is insufficiently designed, it may be necessary to go back to the previous phases to perform a more thorough analysis and adapt the procedural model.

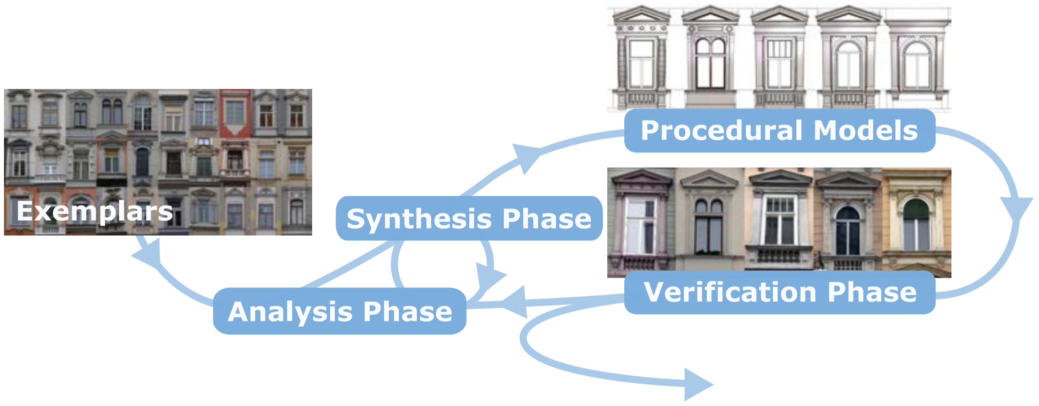


Fig. 5.6 Based on a set of exemplars, the Generative Fact Labeling method first structures observations to create labels and label groups in the analysis phase. The method then proceeds to the synthesis phase in which procedural building blocks are created based on these labels and label groups. This yields procedural models, which are then compared in the verification phase to the set of exemplars to determine whether the desired accuracy has been achieved or not. The whole process is not linear. It may be required to return from the synthesis or verification phase back to the analysis phase to perform a more thorough analysis.

The fact labeling approach is defined very generic, so it can be applied to any domain to create a classification scheme. The main distinguishing property of this approach is the *procedural* view. The observations are grouped in way that they can be directly mapped to procedures that can be used to re-create the observed shapes. Consequently, the quality of the classification is directly proportional to the quality of the results and vice versa.

5.2.2 Window Analysis

The analysis is based on an unordered set of 150 exemplar windows (a selection of which is shown in Figure 5.7). The window exemplars are from neo-classical buildings erected in Graz, Austria, in 1860-1890 (*Gründerzeit*).

Studying Appropriate Literature. At the start of analysis phase, appropriate literature needs to be studied to acquire the right vocabulary in this domain. Possible architectural configurations and styles are covered in literature like [Chi05, Mit92, DJ08]. Additional to the aforementioned books about architecture in general, special window-related books like that from Schulze [Sch08] are a necessity to obtain comprehensive information about possible window element configurations, naming of the individual parts, and how they were composed.

A further very important reference, in particular for windows in Graz, Austria, is the work of Ortwein [OS93]. His work on the German renaissance, consisting of nine volumes, is still used as seminal compendium in building restoration due to the high accuracy in the description of details. Ortwein had much influence in Graz during the 19th century; aside from several churches he also designed buildings in the famous “Sporgasse”.

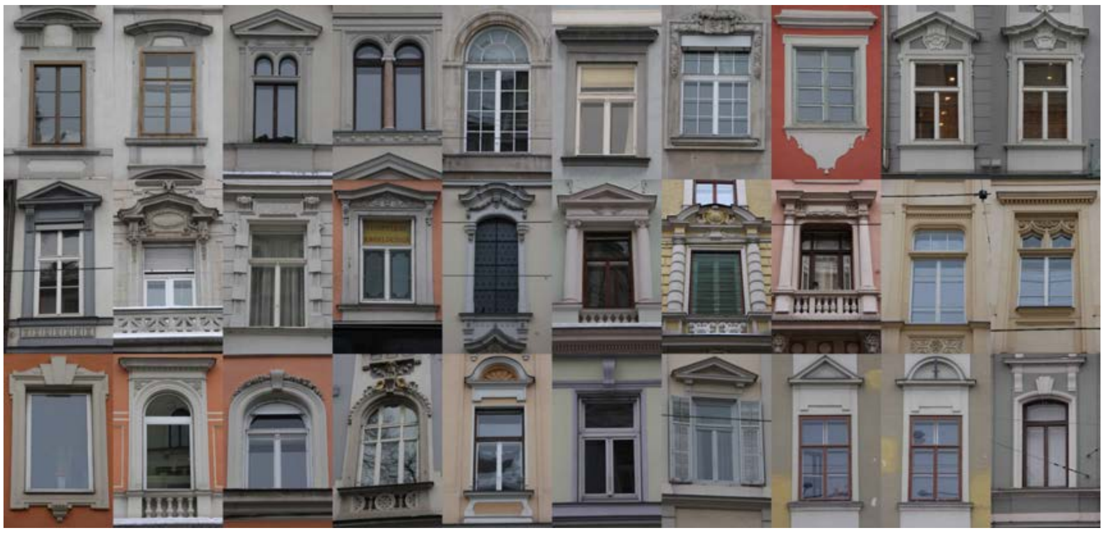


Fig. 5.7 Window exemplars. The full set contains 150 images of windows from neo-classical buildings erected in Graz, Austria, in 1860-1890 (*Gründerzeit*). The complexity and visual dominance of the windows pose challenges to any digital urban reconstruction. (image adapted from Thaller et al. [TZK*13])

It is important to define an abstraction level in the analysis step. Relevant architectural literature is helpful in the communication with domain experts, which leads to more relevant observations. However, broad prior knowledge is not mandatory and may even be misleading. The fact labeling method focuses solely on the geometric aspects of architecture. Experts may say that two windows have nothing in common in terms of architectural history, but they may be labeled the same way due to their similar geometrical features.

Analysis of Windows. The analysis step usually starts with more or less obvious observations such as "this window has a sill" or "next to the window are pilasters". After some iterative refinement of these observations the fact labels and label groups shown in Figure 5.8 (labels for not applicable ($-$) and other ($*$) are omitted) have been attained. Figure 5.9 shows some fact labels on example windows. Each label group, i.e. each line in the table, can be interpreted as a question that can be asked about a window.

A.count How many windows are there?

B.side Is the window framed at the side by columns or pilasters? Alternatively, the decoration above the windows can be symbolically supported by brackets at the *side* of the window.

C.sill Is there a sill below the window, or is there a sill with additional decorations below it?

D.above Is there a cornice above the window, or a pediment, or a combination of the two?

E.frieze Is there additional space, a frieze, or an architrave below that cornice or pediment?

F.layout Do the pillars at the side and the frieze or architrave between the cornice and the opening interact in some way?

G.shape What is the shape of the window opening?

H.frame Is there is an added frame around the opening? Does that frame have a visible keystone at the top?

I.pediment What is the basic shape of the pediment?

J.pediment2 Is there systematic variation of pediment shape, such as extensions to the side?

K.pediment3 Is the pediment open, or is there a keystone?
L.cornice Is the cornice broken in the center?
M.below-cornice Are there brackets that symbolically support the cornice? (This does not include the "side" brackets (B2)).
N.below-sill Are there brackets that symbolically support the window sill?

Some shapes may have similar features which leads to the same questions that are asked in the corresponding label groups. In the example of windows this is true for the application of moldings, which are present throughout different parts of the window.

O.sill2 Is a molding applied to the sill?
P.frieze2 Is a molding applied to the frieze or architrave?
Q.frame2 Is a molding applied to the fame?
R.pediment4 Is a molding applied to the pediment?
S.cornice2 Is a molding applied to the cornice?

L.	Label Group	1	X	2	X	3	X
A	**count**	single window		double window		triple window	
B	**side**	pilaster	f3	big bracket			
C	**sill**	simple sill	a4	sill and decoration below			
D	**above**	cornice	e1	pediment	a1	cornice and pediment	b1
E	**frieze**	frieze/architrave					
F	**layout**	pilasters/brackets beside frieze		pilasters end below architrave		crossing	
G	**shape**	rectangular opening	d2	round arch	e2	segmental arch	a2
H	**frame**	frame		frame with keystone			
I	**pediment**	triangle pediment	b1	round arch pediment		segmental arch pediment	
J	**pediment2**	horizontal cornice at the sides	a1				
K	**pediment3**	open	h1	keystone	g1	stepped	
L	**cornice**	broken	c1	stepped	e1		
M	**below-cornice**	brackets at side	a3	many brackets	c3	centered brackets	b2
N	**below-sill**	brackets at side	c4	balustrade			

Fig. 5.8 This labeling table is the result of the analysis phase. Every label, e.g. A1, is associated with a set of observations on the given facts (exemplars). Entries in the X-columns refer to the table of images of procedural assets in Figure 5.11. (table source: Thaller et al. [TZK*13])

Some questions depend directly on the answers of other questions; if a window has been labeled as not having a cornice or pediment (D−), all other questions about the shape of the pediment and cornice, consequently, have to be answered with "not applicable" as well. These hierarchical relations are obtained from the observations, but the fact labels do not form a strict hierarchy naturally. The relation between the extracted label groups can be quite complex, especially considering how the side decoration interacts with the frieze or architrave (see Figure 5.10).

5.2.3 Window Synthesis

The window analysis must eventually lead to the synthesis of three-dimensional window elements based on the label groups formed in the analysis step. To achieve this, the shapes to be

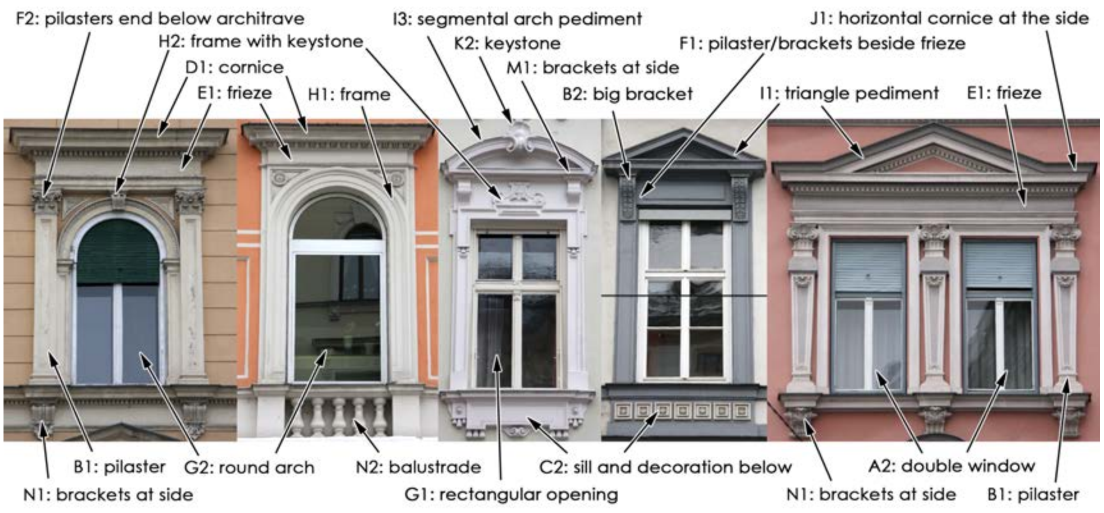


Fig. 5.9 Various fact labels, which are derived from observations, applied to five example windows. The labeling is not intended to be complete for all of these windows. (image source: Posch [Pos13])

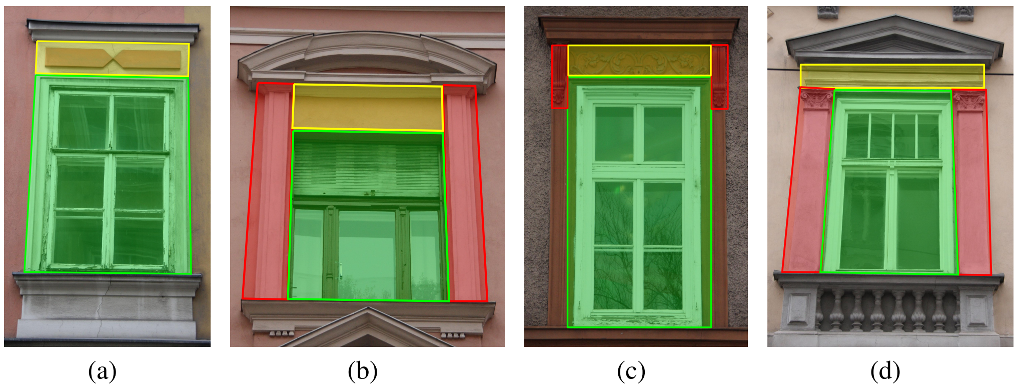


Fig. 5.10 Pilasters, brackets, friezes and architraves, and their possible arrangements. A window with no side decoration but with a frieze (yellow) is shown in (a). Pilasters (red) bypass the frieze in (b). Pilasters can be reduced to smaller brackets that support the cornice (c). Finally, pilasters end at the top of the window opening and support an architrave (d). (image source: Thaller et al. [TZK*13])

produced have to be factored out into re-usable procedures. The realization of these procedural window elements has been done by Posch in his Bachelor's thesis [Pos13].

To illustrate the synthesis process, let us take the pediment window element as an example. Four label groups have been used to describe the pediment shape. These four label groups cover the basic shape, optional extensions to the side, detail variations such as keystones or missing parts, and moldings that are applied to the pediment shape. One single procedural building block that features many different parameters to achieve all these variations is not desirable. Ideally, different operations are factored out, which can then be combined to achieve all these variations. For example, the necessity to apply moldings has been identified in many different window elements. This an indicator that moldings, which are, technically speaking, profile sweeps along certain edges of a shape, need to be realized as a standalone pattern, which can then be re-used in many different places. Even though the detail variations do not occur in as many places as moldings, it is, nevertheless, advantageous to factor them out too. By factoring out such features, the construction of the underlying shape becomes easier. In case of the pediment shape, the last thing that remains is to describe the basic shape, which is triangular or round and may be with our without side extensions. The combination of the operations for the basic shape, the moldings and the details allows to generate a wide variety of different pediment shapes. This procedure is applied to all label groups to find and map re-usable patterns to operations, which can then be used to quickly obtain scripted building blocks for interactive procedural modeling, in this case window elements.

A selection of the realized procedural window elements is illustrated in Figure 5.11, and some of the operations to create these elements are described in the following. An index like (a1) refers to an image in the table of images in Figure 5.11.

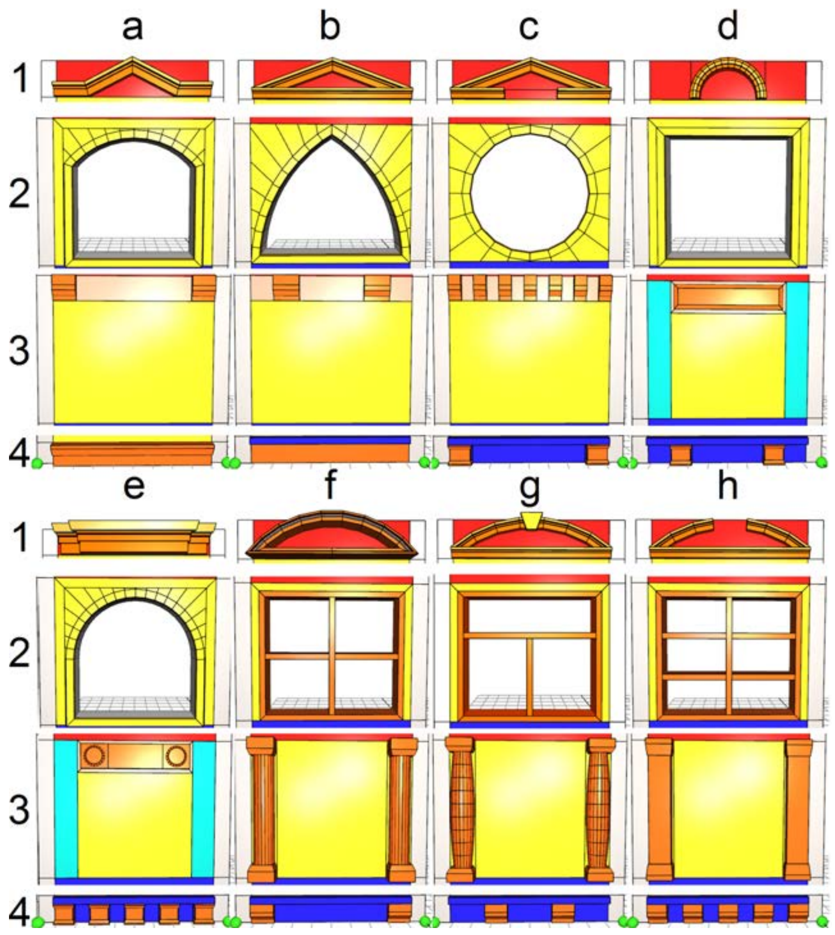


Fig. 5.11 Samples of procedural assets from the window part library: Cornices and pediments (first row, a1-h1), window shapes with borders and crossbars (second row, a2-h2), friezes, panels and pilasters (third row, a3-h3), and window sill and decorations (bottom row, a4-h4). All these assets adapt their size to the space they are inserted to. (image source: Thaller et al. [TZK*13])

Procedural Models of Window Shapes and Crossbars. The most common window hole shapes are supported by the procedural window library. These include the common rectangular shape (d2), round shape (c2), as well as several arches. Among these arch shapes are round arches (e2), segmental arches (a2) and lancet arches (b2). Crossbar assets (f2-h2) are realized independently of the shape of the window itself. The crossbar rules adapt automatically to the (convex) space that is provided for them.

Procedural Frames and Moldings. The library utilizes the frame-split operation mentioned before in Section 4.2.2. This operation is very versatile and is used for frames of the window pane (a2-e2) and to generate the shape of the pediment (b1-d1,f1-h1). Geometry can additionally also be generated along a polyline for special pediment shapes (a1).

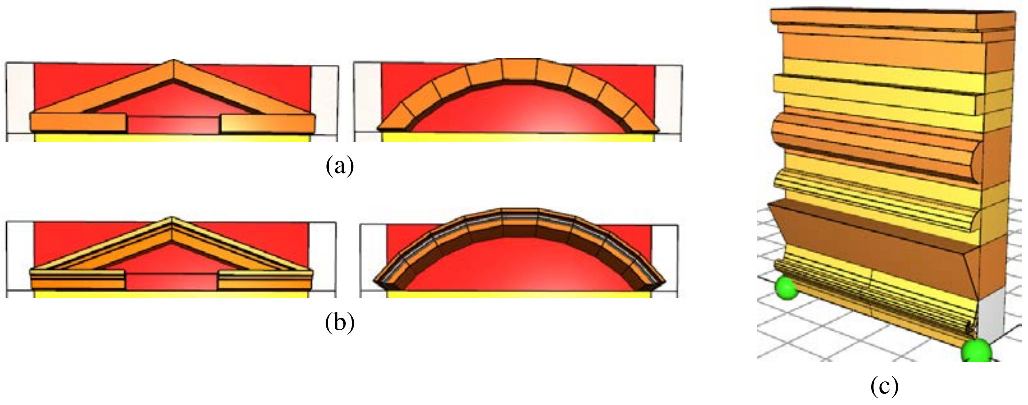


Fig. 5.12 Different moldings applied to assets created with operations through the procedural window library. Pediment and cornice are generated in a generic way (a), the molding is then applied using a specific profile (b). The different moldings (c) are defined independently from the assets that they are applied to. (image source: Thaller et al. [TZK*13])

The output of these operations can be further refined through different moldings (see Figure 5.12). To refine a shape through a molding, an extrusion profile is applied along the course of this shape. This profile is created independently from the asset it is applied to. The cornice and pediment (e1), the architrave, and the window sill (a4) can also be refined by adding moldings to them.

Procedural Models of Cornices and Pediments. Two basic pediment shapes – triangular and round – are supported in the asset library, with the possibility of adding customizations, such as extended end parts (a1), open top sections (h1), the addition of a keystone (g1), or stepped designs (e1). Open pediments are realized using a Boolean difference operation. Stepped designs for pediments, cornices and keystones can be achieved by a separate extrusion step. Keystones, in particular, are inserted by extruding a part of the pediment to the front, as well as up and down. Broken cornices (a1,c1) are supported besides the regular ones. Moldings can be applied to further enhance the appearance of the pediment and the cornice.

Procedural Models of Pilasters and Brackets. Two types of pilasters – a round (f3-g3) and a rectangular pilaster (h3) – are included in the library. Those assets are only equipped with a basic capital and pedestal. Their usage can be very versatile, from being the essential part of the balustrade, to various uses in friezes and other decorative elements.

In most cases, pilasters are exchangeable with brackets. The appearance of brackets can be very detailed and quite diverse, so only a crude approximation is provided in the library to give the basic idea of the real shape. Especially friezes are often decorated with a multitude of brackets (a3-c3). Two fundamental types can be identified, pillar-shaped brackets (c4-e4) for which the pilaster assets are used, and brackets with a slanted bottom part (a3-c3, f4-h4) that only support the element above them.

Procedural Models of Friezes and Architraves. Several decorative elements are placed symmetrically in the frieze (a3-e3). Supporting elements like brackets or decorative panels (d3-e3) are also used within the frieze. Panels can be achieved by combining different operations, such as extrusion, frame and bevel operations. Architraves, on the other hand, are often just decorated with a molding that runs across the width of the window. The before mentioned profiles can be applied on architraves too.

Procedural Models of Window Sills and the Decorations beneath it. Window sills are realized by an extruded part, which can have a molding applied to it (a4). The space below the sill is often decorated like friezes (c4-h4). Brackets then support the sill instead of the cornice. A decoration unique to this space is a balustrade. The pillar assets described before can be utilized here as well. The number of pillars automatically adjusts to the space available.

5.2.4 Window Verification

Finally, the procedural assets created in the synthesis phase can be utilized and combined for reconstruction of windows. The modeling process follows the labeling process in its coarse to fine manner. To some extent it can be seen as a shape grammar because parts are selected and replaced by other new parts. With shape grammars, however, crossings of vertical and horizontal structures are hard to realize, which is sometimes done in the GML Compositor. Thus, a window is a combination of discrete assets, which, although it is done with a graphical user interface, can be seen as interactive scripting.

Although the number of different procedural assets appears to be fairly limited, quite a variety of shapes can be achieved through combining, nesting and repeating the available elements. Through the versatility of the procedural modeling approach it is possible to cover the architectural variety of the given exemplars to a good extent.

Reconstruction of Window Exemplars. Figure 5.13 shows reconstructions of five different windows from the set of exemplars. All of these windows have different layouts to demonstrate the versatility of the generated procedural library. Some decorations are only approximated by manually placing variations of other assets at certain positions.

Based on a finished model, other (similar) windows can be realized quickly by manually adjusting certain dimensions and exchanging a few assets. The benefit of the procedural approach is that the parts can adapt flexibly to a wide variety of surroundings. Therefore, in most cases, it is much more efficient to adapt an existing window than to create a new window from scratch. This fact encourages the re-use and re-parametrization of models.

Figure 5.14 shows some variations of the two windows in red and blue frames from Figure 5.13 realized by just exchanging assets and adjusting parameters. In terms of operations these windows are “close”, which suggests that *procedural distance* could be a useful shape similarity measure.

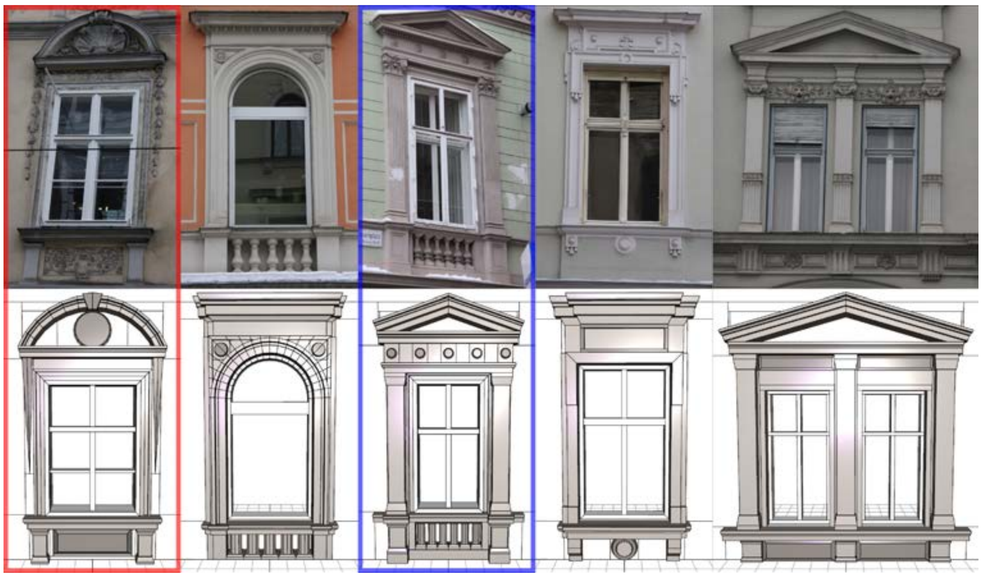


Fig. 5.13 Synthesis of example windows. The first exercise was to reconstruct five window exemplars with sufficiently different layouts and decorations. The second task was the variation of the two windows in the red and blue frames (see Figure 5.14). (image source: Thaller et al. [TZK*13])

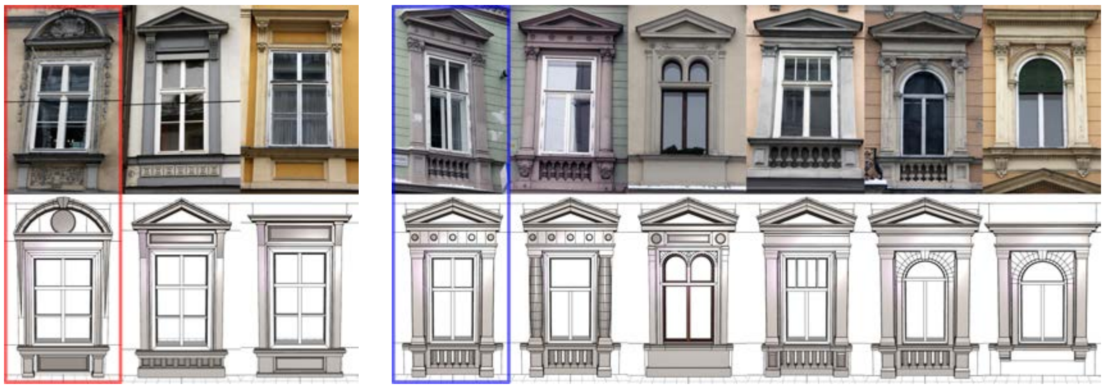


Fig. 5.14 Variations of the two windows in red and blue frames from Figure 5.13. The variations are derived only by replacing assets and manually adjusting proportions. Assets can be combined to realize also more challenging configurations (right group, third variation). (image source: Thaller et al. [TZK*13])

Step-By-Step Modeling of a Window. For the particular window in Figure 5.15 at first it is decided what kind of window layout is used. In this case the central part is realized by partitioning it into four parts: a center part for the window, and two pilaster parts, which support the final frieze part (see Figure 5.15(b)). All measurements and sizes of specific parts can be modified by parameters, i.e., the width of the pilaster parts can be adjusted manually. Further modeling is done by asset insertion steps, which can be executed in any order since each asset operates on a single selected part (see Figures 5.15(c) - (g)). Some assets create new spaces, where further different assets can be inserted. Examples for this are the window opening assets, which allow the insertion of border and crossbar assets to the opening part. Finally, details, such as moldings and keystones, can be further applied to the inserted assets, which leads to the final model (see Figure 5.15(h)).

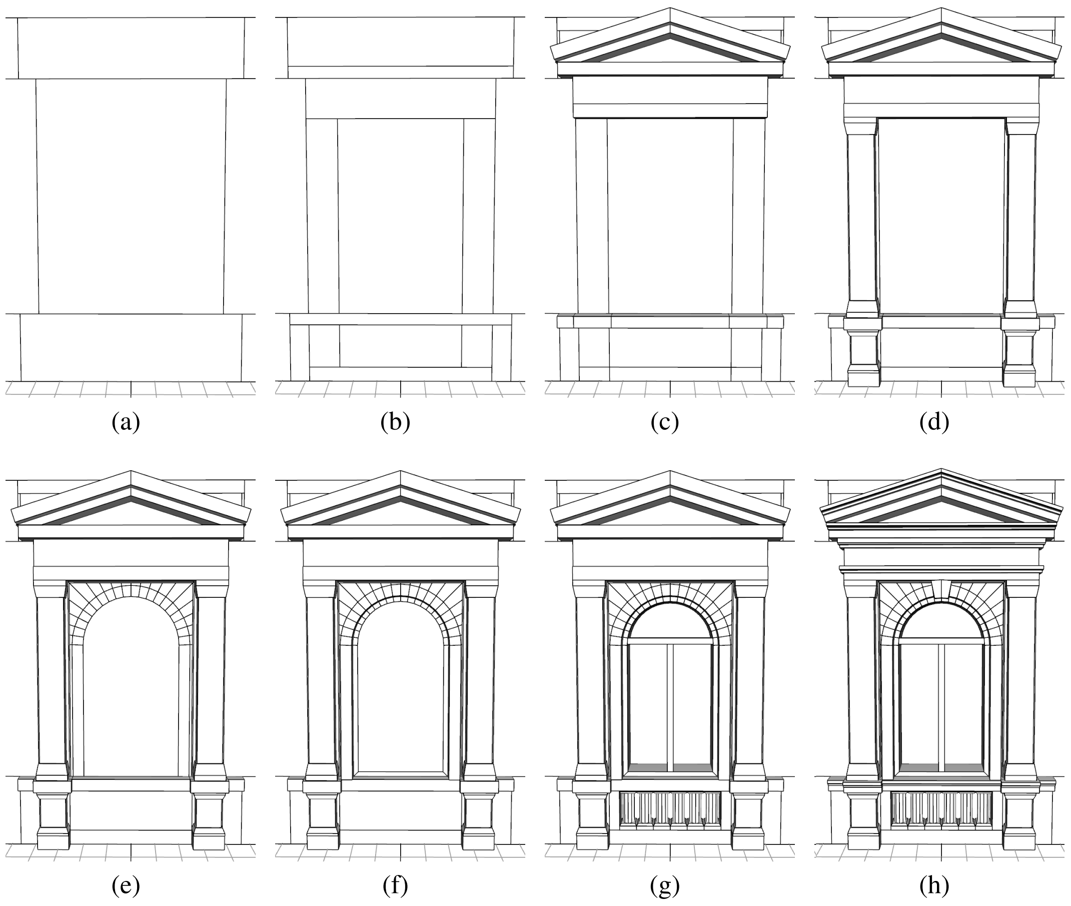


Fig. 5.15 Interactive window modeling using procedural assets. First, the layouts for the individual sections are chosen ((a), (b)), in this case a single window with decoration above and below the hole. The order in the asset insertion steps ((c) - (g)) is not important since assets can be inserted independently from each other. The last step is to apply detail moldings to the window elements and to add the keystone (h). (image adapted from Thaller et al. [TZK*13])

5.2.5 Evaluation of the Generative Fact Labeling Method

With the procedural building blocks that have been synthesized based on the assigned fact labels, more than 80 percent of the acquired windows can be reconstructed in the level of detail shown in Figure 5.14.

In this section the limitations of the Generative Fact Labeling method, in the context of the presented examples on windows, will be discussed in three levels. First, challenging window examples are presented that are hard to synthesize with the current asset library. Afterwards, shortcomings of the classification are discussed followed by whether or not this invalidates the fact labeling approach.

Challenging Window Examples. The windows shown in Figure 5.16 are grouped into a set of difficult but feasible (left) and fundamental problems (right). The left group of windows – which amount to roughly 10 percent of the set of exemplars – all exhibit features that are not yet supported, such as new opening shape assets ((c), (d) and (j)) and frame decorations (b). These features have been unique to single windows and are not so common throughout the analyzed shape space, which is the reason why these features have been left out in the initial analysis. Furthermore, the decorations around windows (a) and (i) are out of scope, since the analysis was limited to convex partitions. Such decorations require a shape analysis approach of their own. Nevertheless, all these missing features have a well-defined place within the classification in Figure 5.8.

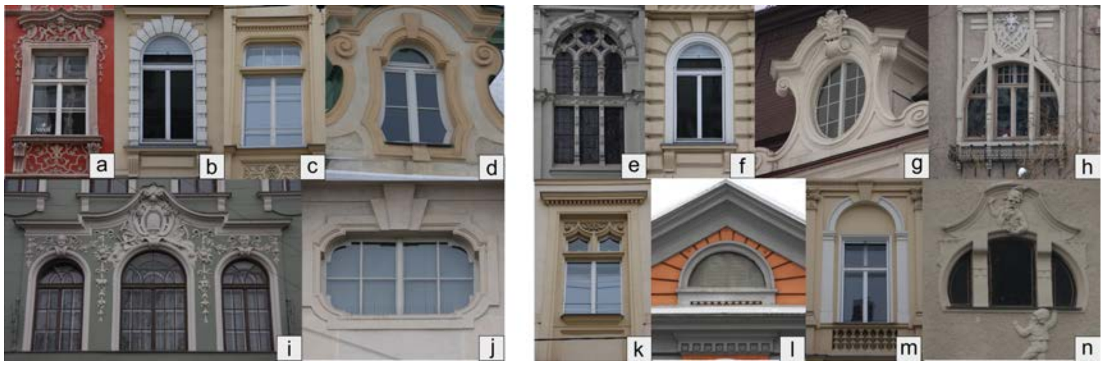


Fig. 5.16 Windows that cannot be handled properly by the extracted procedural library. The left group of windows could be handled by special-purpose assets. The right group reveals more fundamental issues and problems. (image source: Thaller et al. [TZK*13])

The problems of the windows on the right side are more of a fundamental nature. The analysis of these windows led to the classification of the following issues:

General Construction Issue. Windows (e) and (k) exhibit delicate tracery in the top, leading to bar and hole shapes that require specific geometric constructions which are not found elsewhere.

Cross Feature Issue. Window (e) also features a ledge that runs along the façade and crosses the window on a horizontal bar, interacting with various structures of the window along the way.

Cross Hierarchy Issue. Even more drastic is the cross hierarchy issue; the seemingly innocent example is the top of window (m) where the keystone not just protrudes downwards and upwards, but actually bridges and breaks the circular profile, then the horizontal frame, and finally becomes part of the frieze in the top.

Planarity Issue. Window (g) violates one of the implicit assumptions, namely that window decorations are applied to planar façades. The pediment and the structures at the sides of this circular window – which are vaguely reminiscent of ionic columns – are part of the three-dimensional structure of this particular building

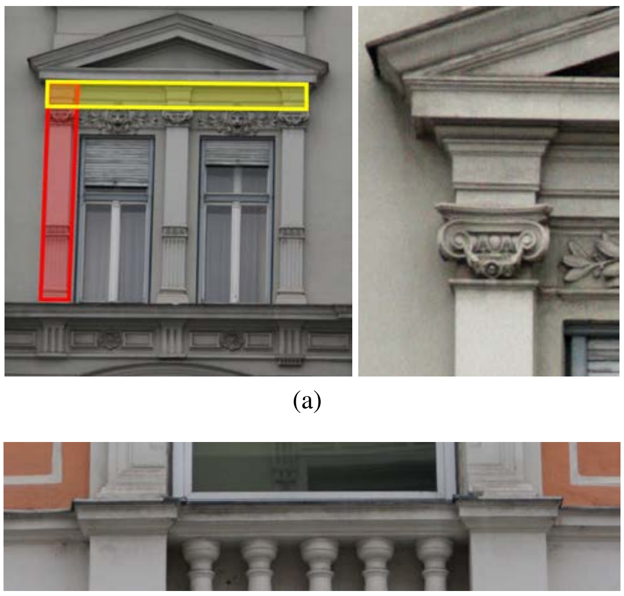
Ambiguity Issue. The holes of the multi-windows (h) and (n) are so tightly coupled that the window classification is ambiguous. Apparently a larger hole was partitioned by bars, so it makes no sense to treat the sub-windows separately; but the bars are also so prominent the windows are clearly separated. There exists so many examples of ambiguities in windows that one might suspect that these ambiguities are introduced intentionally by architects.

Repurpose Issue. The ambiguity issue from before is similar to the repurpose issue of window (l), where the pediment is not on top of the window, but the window is inside a triangle pediment.

Issues with Interacting Parts. By assigning additional labels the fact labeling approach allows making inter-dependencies explicit. For a given window, it is possible to state that there are pilasters that support the cornice and pediment, and there is an architrave below this cornice. This behavior is covered in an extra label group (F) that describes how these two structures interact.

The window in Figure 5.17(a) is labeled D3, as it has a cornice and pediment, and E1 because it has an architrave and frieze. Additionally, it receives the label F3 because both structures overlap. The problem lies in the geometry generation. The assets must be able to deal with the interaction of the architrave and the pilaster. However, this interaction depends on the specific assets that are used for the pilaster and the architrave.

Another example of problematic overlaps are ledges running horizontally over the whole façade (see Figure 5.17(b)). They either do not interfere with the windows, for example between stories, or in other cases they are interrupted by windows. Sometimes, such a ledge is re-used as a window sill in a slightly modified way. The corresponding procedural asset for the window sill would have to describe a window sill created by modifying an existing ledge in a certain way. The focus in this analysis was to limit the attention to individual windows and their immediate surroundings, so this problem for has not been addressed so far.


(a)

(b)

Fig. 5.17 Examples of windows with overlapping structures. The pilaster (red) overlaps the architrave (yellow). In the overlapping area, the molding of the architrave runs across a continuation of the capital of the pilaster (a). Other overlaps occur when a ledge that runs across the whole façade also serves the purpose of a window sill (b). (image source: Thaller et al. [TZK*13])

Feasibility of the Generative Fact Labeling Approach. Due to the problems mentioned, the feasibility of the overall approach is necessary. The question arises whether the method will ever converge and if there is a realistic chance to obtain a reasonably small procedural function library in the end. This library should be able to synthesize a three-dimensional model with satisfactory detail resolution for all exemplars.

A flat list of labels without any hierarchy or other structure emerges from the presented method. Due to the fact that no a priori assumptions are made, the resulting labels can be arbitrarily unorganized. Expectation is that the structural information will be introduced during the exercise. Over-specialization is a great danger when a more detailed analysis of the exemplars produces an ever-growing number of observations, labels, and label groups. This implies that a continuous re-iteration is necessary to identify similarities between labels that can be merged and mapped to the same generative procedure. This inductive reasoning process is called *label reduction*. To illustrate this procedure consider the labels M (below-cornice) and N (below-sill). They reveal a lot of similarities and both labels certainly share most of their procedures and can therefore be merged into a single label.

To reduce this complexity arising by the relation between the individual labels, it is intuitive to follow an approach that decouples the different labels as much as possible. Label dependency is minimal when every label depends only on a single other label. This leads to a linear label refinement process and thus, eventually to a (context-free) parametric shape grammar.

Synopsis

Before the results that are acquired through the presented procedural methods are presented, I wanted take the chance to talk about the analysis required to achieve a procedural representation of given shapes.

We designed a methodological approach to capture a large number of highly structured, similar but not identical shapes. Based on the motivating example of reconstruction the Louvre in Paris, I extended our analysis of a collection of different window shapes in this chapter. This so-called Generative Face Labeling method is composed of three phases: an analysis phase, where a sufficiently large collection of exemplars are analyzed and structured by assigning fact labels, a synthesis phase, in which procedural assets are created based on the preceding analysis, and finally a verification phase, in which the expressiveness of the acquired building block is assessed by comparing reconstructed shapes with the original exemplars.

In this chapter this method was applied to a collection of 150 photographs of windows in the city of Graz, Austria. The application of our approach led to a library of assets, which can be used to reconstruct more than 80 percent of these windows to a sufficiently large level of detail. The remaining 20 percent of windows do either feature shapes that require additional assets, or are unique in the sense that our basic assumptions on windows are violated by those exemplars.

In essence, the Generative Face Labeling method is a generic approach to produce a library of functions that allow reconstruction of given exemplars. It is, however, important that our method should only be seen as a guideline to find at least one reasonable procedural explanation of a complex shape space.

Chapter 6
Applications and Results

Contents

Abstract. So far only methods have been discussed theoretically and only illustrative examples have been presented. This chapter demonstrates applications and results of the applied research done with the technologies presented in the two preceding chapters. All applications that will be discussed in this chapter have been realized using either the GML directly or by using the GML Compositor, which creates GML code in the background. The examples presented in the context of the GML mainly focus on the extension of the state of the art of split grammars. Furthermore, the procedural design of wedding rings – an application close to industry – will be discussed. The GML Compositor has been utilized for procedural modeling in domains of monument reconstruction, Cultural Heritage and Ambient Assisted Living. Furthermore, related results of students that have been supervised by our research group are presented in this chapter too.

6.1 Procedural Design of Wedding Rings

People like their belongings personalized and customized. This demands a paradigm shift in industrial design. Designers do no longer have to create a single static design for a product, but a design for a whole product family. The basis for mass customization is done in modern *computerized numerical control (CNC)* machinery, which allows a variation of parameters for every single produced item. The trend, however, goes in direction of rapid prototyping through the use of three-dimensional printers.

The big obstacle in this domain is the verification of the design. Especially in industry every item needs to be checked for its functionality, durability, manufacturability, aesthetics, and production cost. The parameter space of the procedural description must therefore be defined very carefully, which can lead to very intricate problems for complex products. Therefore, to create a procedural description it has proven very useful to start from a population of valid items, as already discussed in a use case of windows in Chapter 5.

These demands are especially true for wedding rings. For many people the wedding day is one the most important days in their lives. For this event couples often choose a unique individual wedding ring design, which are realized by various workshops where a professional ring designer guides through the whole production process. This makes wedding rings an ideal case for industrial mass customization.

Based on a population of real rings (see Figure 6.1) that had been provided by the German ring manufacturer JohannKaiser [JK13], a procedural description of these was generated similar to the presented Generative Fact Labeling (see Section 5.2.1). The results of this project have been utilized by JohannKaiser ring manufacturing in an online editor that is used by professional ring designers in counseling interviews with customers. This piece of software has been awarded several times and had serious impact on this particular business and showed that there is a demand for procedural editors in industry.

This section gives an overview on the procedural ring design task performed at our institute. This task started with the work of Berndt et al. [BSK*12], which has been extended by me afterwards. For this task, Berndt was responsible for the main parametrization, Schinko for the displacement mapping, Settgast for providing appropriate materials and Krispel for the generation of procedural gemstones. My contribution in this collaboration was providing an extension to the procedural library to support Boolean operations, so that the capabilities and expressiveness of the procedural wedding rings can be increased further.

6.1.1 Ring Features

At first, characteristic elements of ring design need to be analyzed from literature and the provided samples. A ring features:

Profiles.
Profiles are the most dominant feature that defines the appearance and shape of rings. The classic profiles are:

- **Flat section profiles** are the traditional wedding ring profiles, which is flat on the in- and outside.
- **D-shaped profiles** have a flat surface on the inside and a heavily bent (half circle) profile on the outside.
- **Halo profiles** feature a perfectly round cross section.

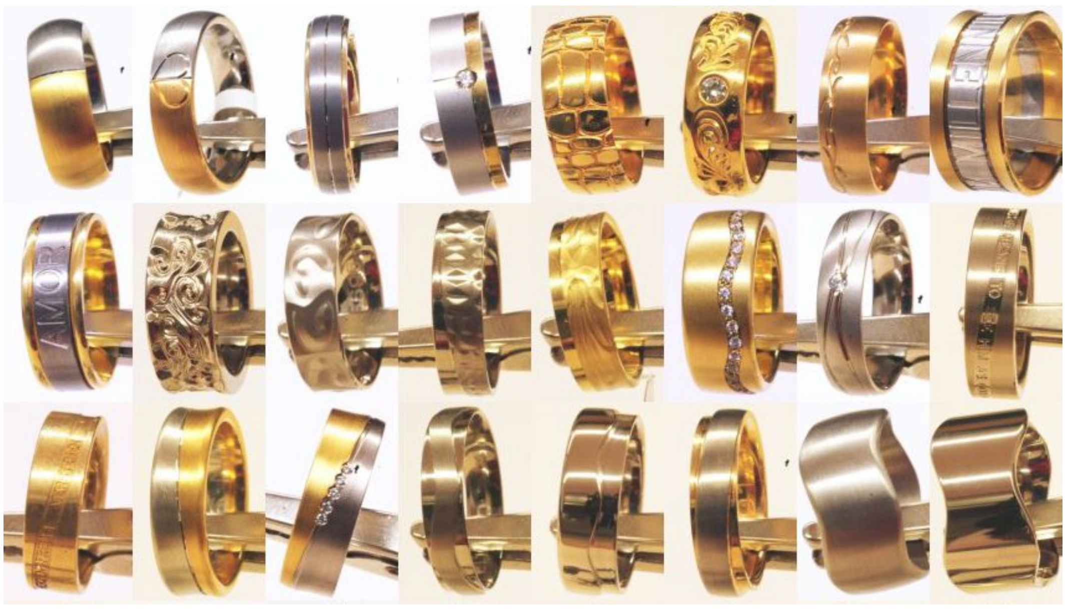


Fig. 6.1 Some of the ring samples from the JohannKaiser [JK13] wedding ring design space. The whole set of 40 rings was used for the creation of a procedural description of weddings rings, which is both an abstraction as well as a generalization of the given individual designs. (image source: Berndt et al. [BSK*12])

- **Oxford court profiles** feature an oval profile with flat rounded internal and external facets.

Materials.
Another important characteristic of a ring is its material. The most commonly used materials include noble metals, like rose gold, white gold, silver, and platinum, but also stainless steel. A ring may feature also multiple different materials, but also the appearance of single materials can be varied through the finish of the surface. The surface finish can range from a polished, highly specular surface, over glossy, to abraded or brushed surfaces with a matte appearance.

Patterns and Engravings.
Rings can be decorated through the use of both engravings and surface patterns. Most wedding rings feature engravings in form of text, but engravings can also realize symbols and other artistic strokes. Patterns can be used to give the ring a more rough and defined appearance. Furthermore, pieces of the ring can be left or cut out to achieve a more unique look.

Gems.
Typically gemstones used in wedding rings are diamonds, but sapphires, rubies, emeralds, amethysts and aquamarines are also commonly used. Gems do not only affect the value of the ring, different distributions of different sized gems can also have their own meaning.

6.1.2 Ring Parametrization and Modeling Operations

The parameters as well as the modeling vocabulary that has been extracted in the analysis phase are presented in this section. Examples of finished ring models are provided to illustrate the different features.

Ring Parameters. Based on a careful analysis of the collection of rings (see Figure 6.1), following parameters have been extracted to define the base shape of most of the rings:

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

By decomposing the design variations into a set of profile transformation functions, the effects of each function can be combined by simple sequential execution to achieve various designs. These transformations can all feature individual parameters; the sine transformation used in Figure 6.2(f), for example, requires three parameters: frequency, amplitude and the points of the profile polygon to which it is applied to.

Step-by-Step Ring Generation. A step-by-step generation of a basic ring shape is illustrated in Figure 6.2. First, a basic profile polygon of the ring is designed (see Figure 6.2(a)) and placed radially several times around the center of the ring (see Figure 6.2(b)). The distance to to center is defined by the radius of the ring. This covers the first three parameters of the ring parametrization. By using the GML operator bridgerings these profile polygons can be connected to form a control polygon (see Figure 6.2(c)) of a subdivision surface (see Figure 6.2(d)). By varying the sharpness of the edges of the control polygon different ring shapes can be achieved. At this step the vertex-based profile transformation, which is the fourth parameter, is set to an identity transformation (see Figure 6.2(e)). By changing it to a selective sine transformation (see Figure 6.2(f)), which affects only the topmost points of the profile polygon, a decorative wave on top of the ring's surface can be achieved (see Figure 6.2(g)). Finally, materials can be set to different parts to finish the look of this ring (see Figure 6.2(h)).

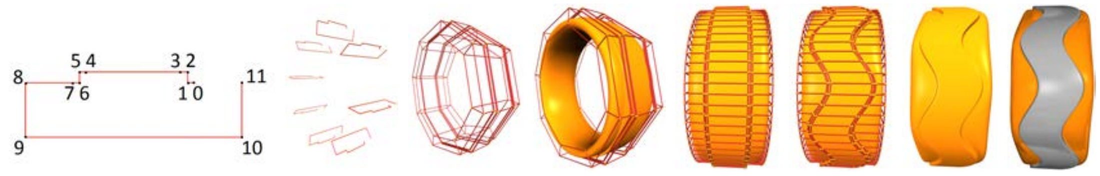


Fig. 6.2 Parametric weeding ring construction begins with designing a profile (a), which is then radially placed several times around the center of the ring (b). These profile polygons are then connected to form a control polygon (c) of a subdivision surface (d). Instead of a identity transformation (e) the profile transformation function can be changed to a selective sine function (f), which creates a decorative wave on the surface of the ring (g). Finally, materials can be applied to the different parts (h). (image source: Berndt et al. [BSK*12])

Patterns and Engravings. Patterns and engravings are applied to a basic ring shape through the technique of displacement mapping. Displacement mapping can be realized on per-vertex and per-pixel basis. The latter only affects the resulting image through changing the normal vectors and thus the lighting calculation. The vertices of the mesh are not affected. The generated rings are not only created for visualization purposes, but also for rapid-prototyping purposes (e.g. three-dimensional printing). For this, actual geometry is necessary and therefore the per-vertex approach, where the geometry of the ring itself is adapted, is the only alternative.

For per-vertex displacement mapping sample points on the surface are taken and displaced in direction of their normal vector. The distance used for the displacement is provided in a separate height map. Displaced points also feature new normal vectors, which are calculated by combining the actual surface normal with one provided in a separate normal map. On the fly a micro-tessellation is created in regions with a surface displacement. The displacement mapping is done entirely on the GPU to provide efficiency. This method is very flexible and allows even modeling of irregular shape features that are too cumbersome to model on the control mesh level. Examples for rings with engravings and surface patterns are shown in Figure 6.3.

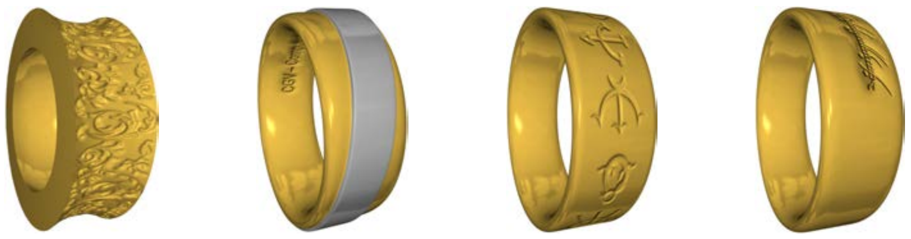


Fig. 6.3 Engravings and surface patterns are realized through vertex-based displacement mapping done on the GPU. (image source: Berndt et al. [BSK*12])

For special surface features like holes, sockets or cut-aways, displacement mapping is not the right tool. Through the use of Boolean operations through Constructive Solid Geometry (see Section 3.7) complex surface features can be realized. Constructive Solid Geometry can be achieved by screen-space methods and object-space methods. Again, screen-space methods are not suitable for rapid-prototyping purposes, but necessary for providing quick previews. Therefore, screen-space methods are used for previewing purposes and an exporter is provided to calculate the real mesh. Examples for rings created through the use of Boolean operations are shown in Figure 6.4.

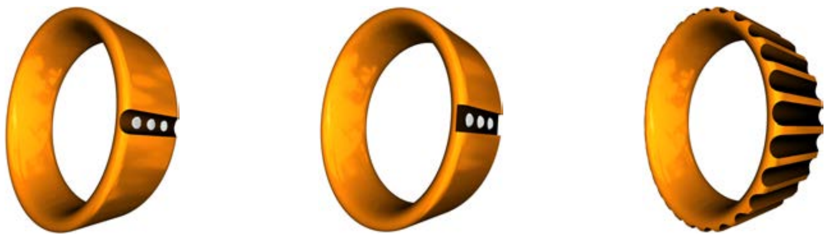


Fig. 6.4 Boolean operations can be used for gem sockets, holes and cut-aways. Those surface features are hard or not possible to realize with displacement mapping.

Placing of Gemstones. The gemstone model used for the rings is the brilliant cut type. Like in the work of Hemphill et al. [HRJS98] gemstones are realized through convex polyhedra (see Section 3.5). In comparison to Hemphill et al., the parametrization of the gemstones used for the procedural wedding rings is slightly simplified. Procedural instances of gemstones can be placed on rings by specifying the apex point, an up-vector, and a scale factor. The position of the gem is determined by a ray-cast. In Figure 6.5 variations of gemstone placements are shown in relation to a real ring.

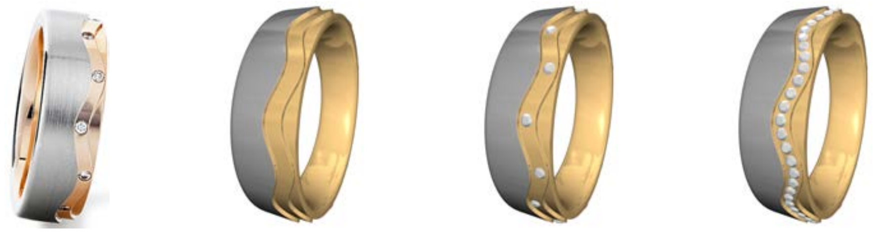


Fig. 6.5 Variations of gemstone placement based on a real ring. Gemstone can be placed procedurally based on surface features of the ring. (image source: Berndt et al. [BSK*12])

6.2 Procedural Monument Reconstruction

Being able to reconstruct monuments through generalized techniques is an important feature for a reconstruction technique. Famous monuments are very special and are usually of non-trivial size and complexity. If a reconstruction technique is able to process such unique examples of architecture it furthermore proves that it is suited to process regular architecture and structures as well. This underlines the importance of the results presented in this section.

These results include the interactive reconstruction of the Louvre through the GML Compositor (see Section 4.4), which started in Chapter 5. In the context of split grammars on convex polyhedra, which have been discussed in Section 4.2, procedural models of the famous Rialto Bridge in Venice and the Eiffel Tower in Paris have been created. These two models have not been generated through an interactive tool, but they offer parameters that allow the creation of variations of these iconic architectural monuments.

6.2.1 Interactive Reconstruction of the Louvre

This section focuses on interactive reconstruction of the Louvre accomplished with modeling tool that is developed at our institute, the GML Compositor (see Section 4.4). The procedural window building block library that has been generating in the case study in 5.2 has become an integral part of modeling tools for monument reconstruction featured within the GML Compositor.

The partial reconstruction of the Louvre presented in this section was done in our work [ZKT*14]. The modeling operations GML Compositor had been utilized by the student Pszeida in context of his Bachelor's thesis [Psz14] to assess the practical applicability and usability of the GML Compositor prototype. His input was very important to improve the handling of the system. He was able to approach the task from an end-users perspective, except that the prospective end-user will have a background in art history rather than computer science. This section details the reconstruction progress and shows all results that have been achieved up until now.

The reconstruction features several façades of the Louvre highlighted in the ground layout shown in Figure 6.6. These include

- façades of the Cour Carée,
- the arcade section of the Cour Napoléon, and
- Perrault's Colonnade, the outer façade of the east wing.

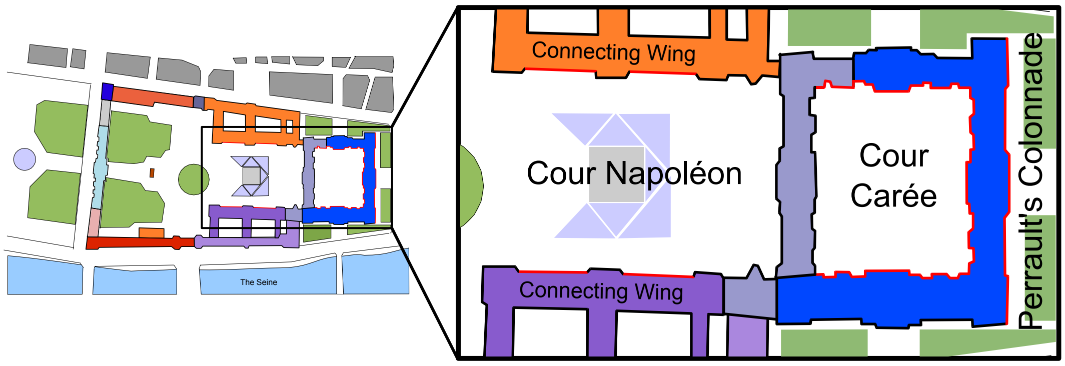


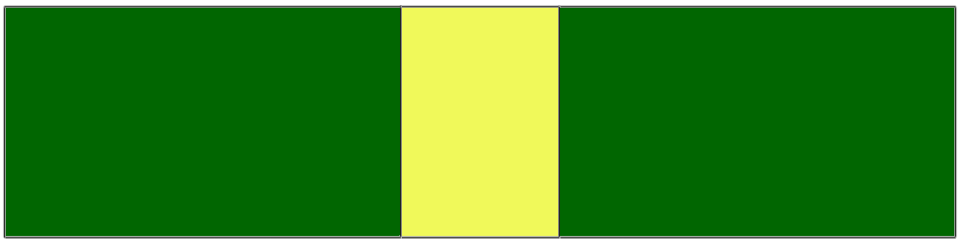
Fig. 6.6 Ground layout plan of the Louvre. The reconstruction covers the façades highlighted in red. These are façades of the Cour Carée, the arcade section of the connecting wings of the Cour Napoléon, and Perrault's Colonnade, the outer façade of the east wing. (original image copyright by Arnaud Gaillard, license: CC BY-SA 3.0)

The Reconstruction of the Cour Carée. Based on the available source material, roughly 80 percent of the façades of Cour Carée could be reconstructed so far. Small differences in the three reconstructed façades have been neglected for the sake of a more efficient reconstruction. Figures 6.7(a) - (c) show important steps in the split hierarchy of the façade. Different colored spaces indicate linked groups. The obtained level of detail of the final model is shown in Figure 6.7(d).

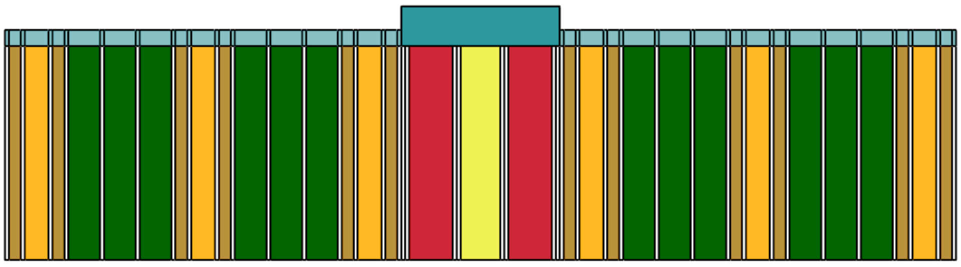
Naturally, the speed of the reconstruction depends on the degree of acquaintance with the modeling system. After some time to get used to the system, Pszeida was able to reconstruct the façade of the Cour Carrée in roughly twelve hours with a preceding ten hours of planing. A coarse approximation was achieved faster, in a few hours, and the remaining time was spent modeling finer details. A rendering of the façades from the Cour Carée is seen in Figure 6.8.

The Reconstruction of the Arcades of the Connecting Wings in the Cour Napoléon. As for the Cour Napoléon, about 40 percent of the façades are finished. For the remaining parts a coarse façade layout has been reconstructed. The arcade section of the Cour Napoléon was done in about nine hours after four hours of planing.

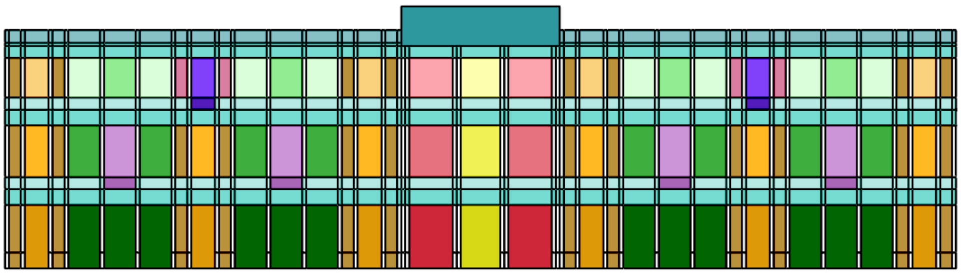
To achieve real three-dimensional models, façades that are reconstructed with the system can feature more than just extrusions in one direction. Through operations that can be applied in any of the three major axis directions, more complicated three-dimensional structures can be achieved as well. The cross-vaults in the arcades are achieved by a combination of two applications of an arch asset operation (see Figure 6.9).



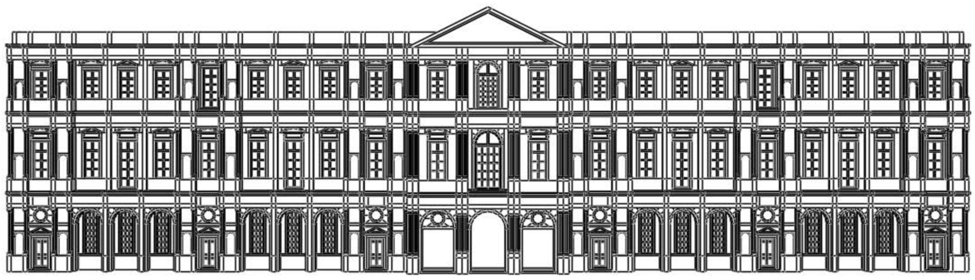
(a)



(b)



(c)



(d)

Fig. 6.7 Important steps in the hierarchical decomposition of a façade at the Cour Carée. Same colored spaces indicate linked groups. First, the symmetry of the facade is achieved (a), then the horizontal (b) and vertical (c) splits are done. Spaces colored in purple color tones indicate exceptions in repetitions. At the end details are inserted leading to final level of detail (d). (image adapted from Zmugg et al. [ZKT*14])

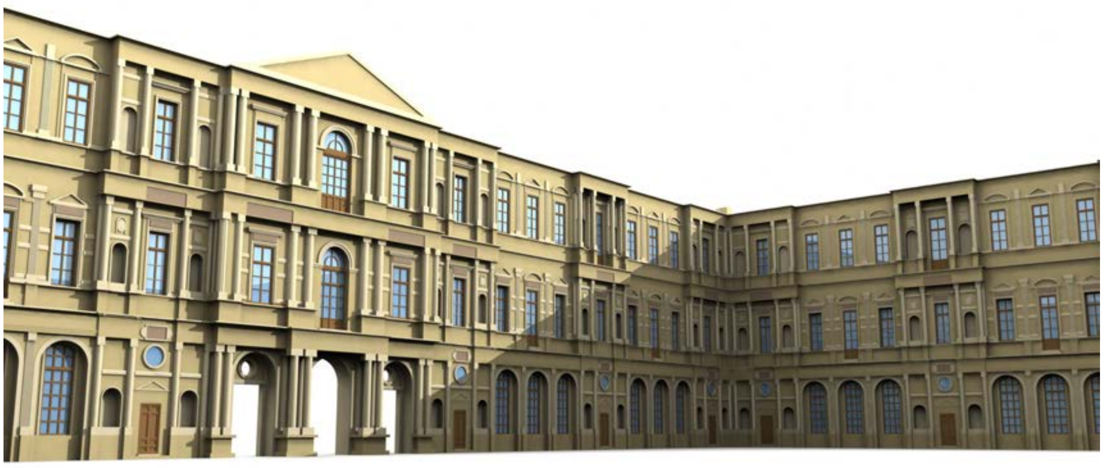


Fig. 6.8 Rendering of the current model of the Cour Carrée reconstruction. (image source: Pszeida [Psz14])

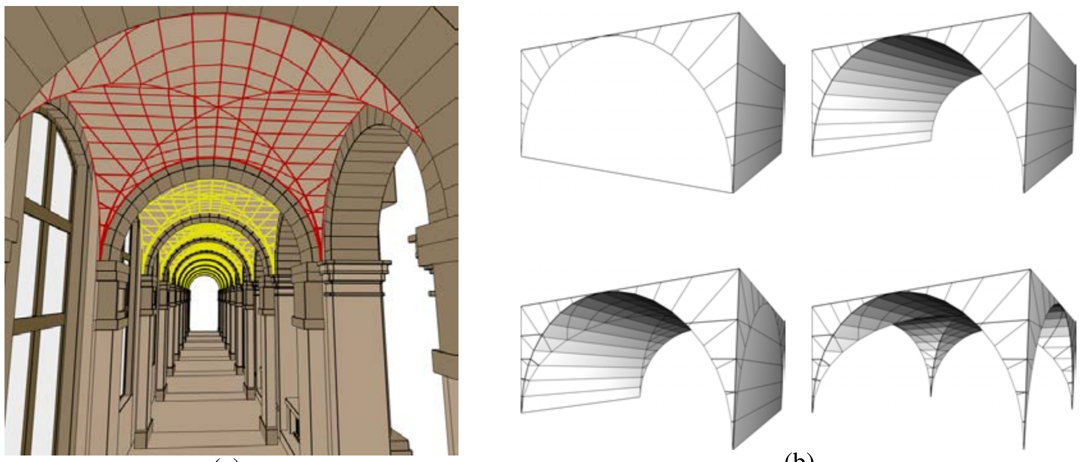


Fig. 6.9 The cross-vaults (a) can be achieved through successive application of an arch asset operation in different directions (b).

To judge the obtained level of detail a comparison between photograph and model is shown in Figure 6.10. These images show that still some work is needed to achieve the same level of detail, but that parametric model can be further developed once appropriate measurements and more detailed photographs or plans become available. In the case in which textures are not sufficient for representing the fine surface ornaments, a separate analysis is necessary to describe them procedurally.

The Reconstruction of Perrault's Colonnade. The reconstruction of the pavilion part of Perrault's Colonnade was discussed in detail in Section 5.1.2. It took about eight hours for Krispel and me to reconstruct the pavilion. A rendering with emphasis on interesting parts is shown in Figure 6.11.

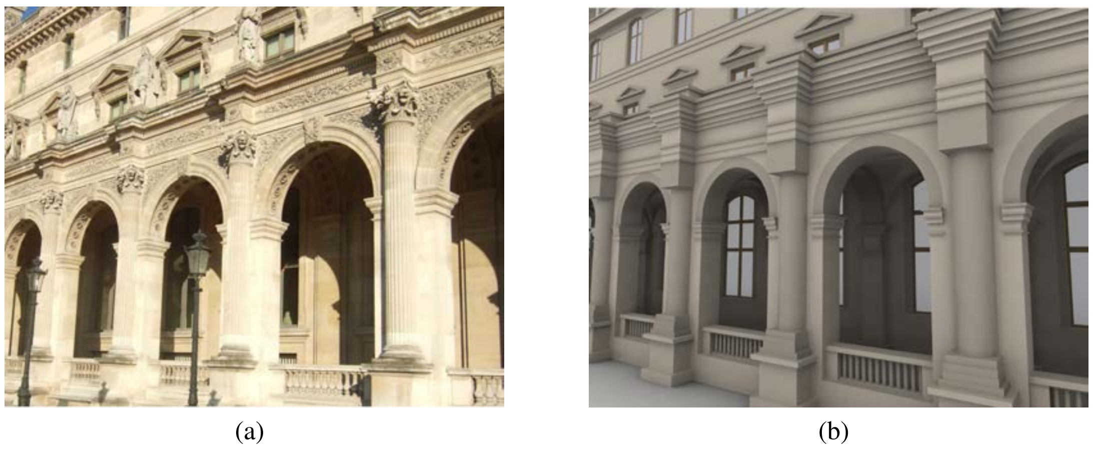


Fig. 6.10 Comparison between a photograph of the arcade at the Cour Napoléon and the current reconstructed version. (image source: Zmugg et al. [ZKT*14])

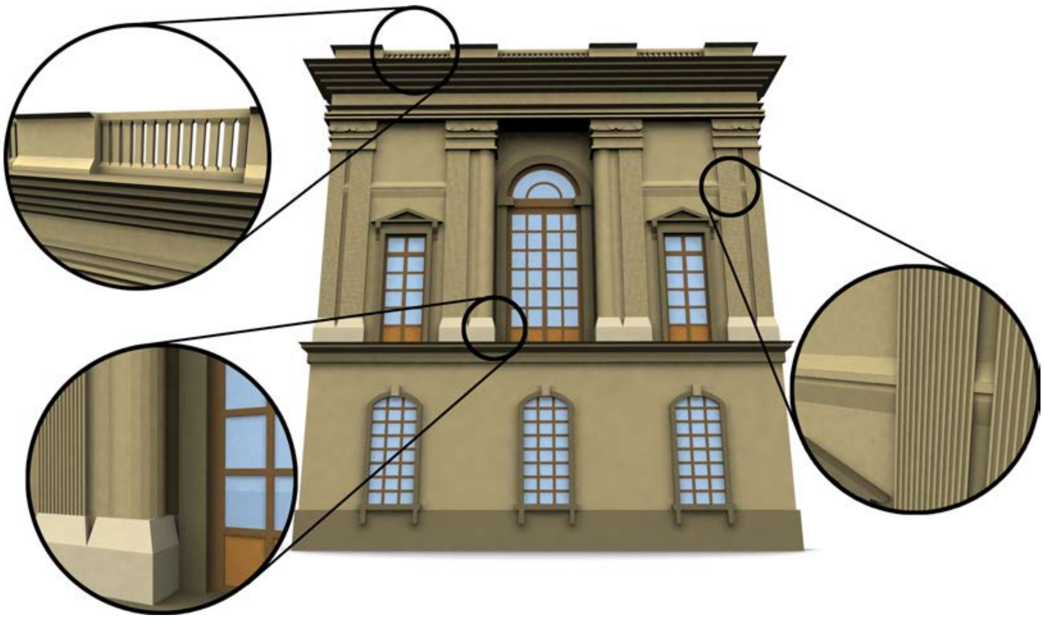


Fig. 6.11 Rendering of the final step of the reconstruction. Special parts such as the round pillars, the parapet and linked ledges are highlighted. (image source: Zmugg et al. [ZKT*14])

The whole façade containing the pavilion was reconstructed by Pszeida independently. He needed about twelve hours with additional ten hours for planing to reconstruct the whole façade. This façade is a little bit less detailed than the separate pavilion. Figure 6.12 shows the acquired result.

6.2.2 A Procedural Model of the Rialto Bridge

In course of the research on split grammars on convex polyhedra (see Section 4.2) I attempted a procedural reconstruction of the famous Rialto Bridge in Venice (see Figure 6.13) because its

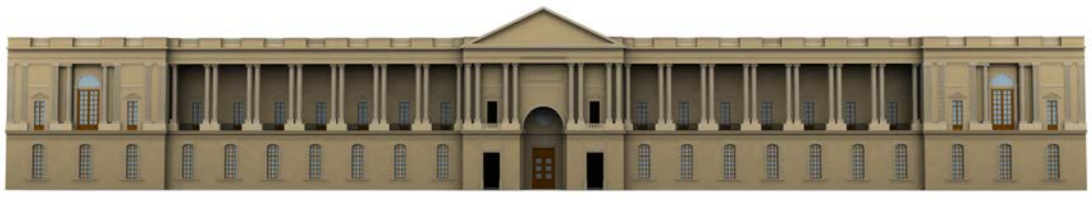


Fig. 6.12 Rendering of the whole east wing façade, containing the pavilions, the wings and the vestibule in the middle. (image source: Zmugg et al. [ZKT*14])

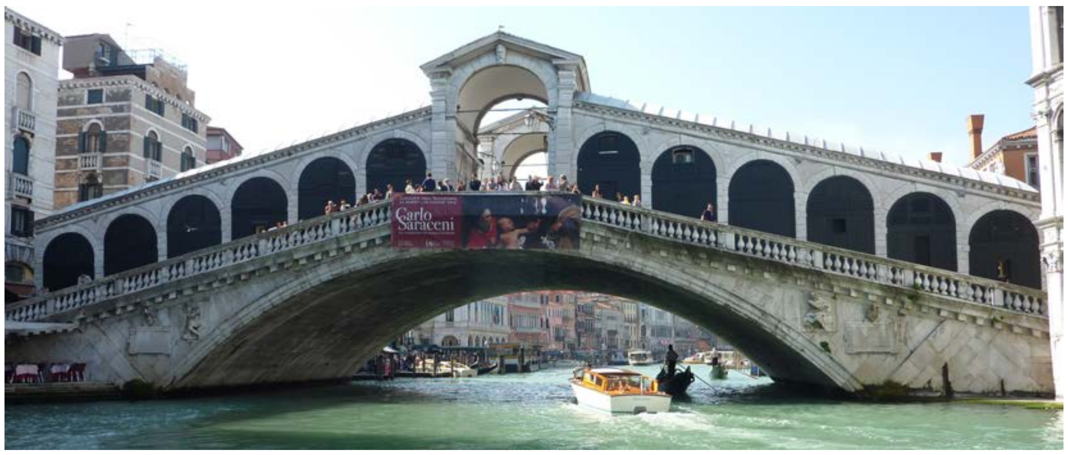


Fig. 6.13 The famous Rialto pedestrian bridge in Venice, Italy. (photograph courtesy of Pia Niederdorfer and Michael Schwarz)

design demonstrates the limitation of box grammars and shows the advantages of using convex polyhedra instead. These advantages of convex polyhedra are demonstrated in the following ways:

- The bridge's slope defines the shape of the basic scopes that form the bridge. These scopes are then split without involving the slope's angle in any calculations. Further operations take the shape of the scopes into account and no modifications need to be made for specific slopes.
- All arches used in this example adapt to their scope. No pre-modeled assets have been used and scaled to match proportions. Due to this fact all arches are always circle segments and are never distorted to ellipses.

The detailed explanation of the procedural reconstruction of the Rialto Bridge performed in this section extends the explanation that has been previously done in our work [TKZ*13b]. After analyzing the structure of the bridge, this section will demonstrate how the procedural split grammar model was realized.

Bridge Analysis. The symmetric Rialto pedestrian bridge spans over the Canal Grande and connects two city parts of Venice. The most important parameters are visualized in top, front and side view in Figure 6.14. The bridge has a length l of 48 meters, a width w of 22 meters, and the passageway has a height p of 7.50 meters. Further measurements have not been available to us, so all relations between the sizes of parts are based upon available photographs. Through the procedural reconstruction of the bridge, however, real measurements can be incorporated

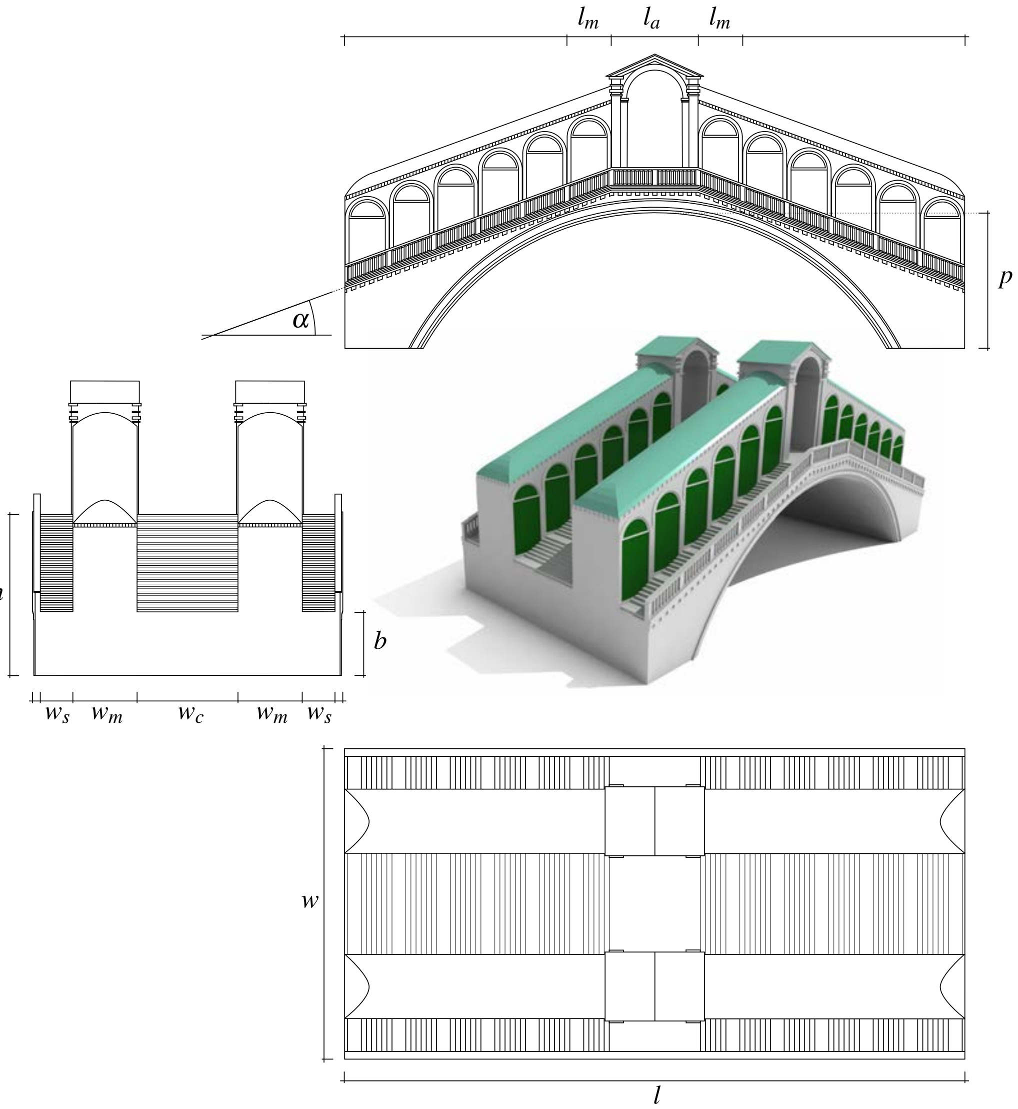


Fig. 6.14 Top, front, and side view from a model of the Rialto Bridge depicting the important basic parameters. Besides the width w, the length l, and the height h of the bridge there is the height of the passageway p, the base height at the sides b, and the angle of the slope α. Detailed measurements include the width of the stairs w_s and w_c, the length of the market stalls w_m, the width of the market stalls l_m, and the width of the central gate l_a.

at any time. Characteristic for the basic shape of the bridge is a slope with an angle α from the sides with a base height b to the middle part with height h to reach an appropriate height p for boats to pass through below.

The bridge features two characteristic lines of inward-facing market stalls on the slopes. In total there are 24 shops with length w_m and width l_m with six in each of the four distinct parts. The six shops that belong together share a round roof and are placed along the slope. The space for each stall is defined through a sheared box in which an arch-shaped entrance is inscribed.

Between these two lines of market stalls, as well as behind, are stairs for pedestrians to overcome the slope. The width of the stairs in between the shops w_c is bigger than the width of the stairs behind the shops w_s. In regular distances these stairs feature longer even parts to ease the entrance to the shops.

At the peak of the bridge there is an even area without any shops. Instead of the shops there are two gates connecting the three different paths to cross the bridge. The height of these archways is bigger than the height of the part containing the market stalls. Furthermore, the archways have two pillars that are symbolically supporting the separate triangle-shaped roof. The arches are inscribed in the convex shape defined by the straight pillars on the side and the triangle-shaped roof at the top.

The balustrade on both sides of the bridge features one segment containing nine vertical pillars for each market stall and two segments of nine vertical pillars for the central gates. This indicates that the width of the archway part l_a in the middle is twice as wide as the width of single market stalls l_m.

The passageway is realized through a circular segmental arch and is inscribed the the convex shape defined by the two slopes and the even mid part at the peak of the bridge.

The bridge, furthermore, features a lot of decoration. All borders of arches, which include the passageway, the archway at the top, and the market stalls, are decorated. The archway, additionally, features a face-shaped keystone decoration. A decorative line is also placed above the market stalls and runs across the whole part to which the shops belong. Additionally, the balustrade features a decoration on the side facing the river.

Reconstruction of the Bridge. Through the parameters length l, width w, height h, base height b, height of the passageway p, and the slope angle α the shape of the bridge is clearly over-determined. There are many different parametrizations that express certain parameters through others. Which parameters are directly accessible depends on the application. For the reconstruction of the bridge I decided that the parameters b and p and are indirectly expressed by the parameters l, h and α. Alternatively, the parameters p and α can also be defined by the parameters h, l, and b.

Although variations of the partitions along the width and length of the bridge and their relations to each other is easily possible, I decided to keep true to the original bridge because I did not want to alienate it too much. Therefore, the model is designed to keep the relations $w_s : w_m : w_c$ and $l_m : l_a$ as close as possible to the corresponding relations of the original bridge.

Decorations of the bridge are approximated through split grammar rules featuring extrusions. All these decorations adapt to all size changes without distortions because no detail has loaded from external meshes.

Varying the width w of the procedural bridge model introduces further lines of market stalls and gates. The width of the market stall line w_m and the width of the outer stair parts w_s are kept the same and the width of the stairs in between w_c varies to ensure a continuous transition. Varying the length l introduces further market stalls in all lines while keeping the width of the gate the same. Varying the height h indirectly also changes the base height b, which increases the space below the bridge. Finally, by varying the angle α different slopes, to which all elements of the bridge adapt automatically, can be achieved. Variations of w and l are shown in Figure 6.15 and variations of h and α are shown in Figure 6.16.

An intermediate step of the bridge grammar (as seen in the first row of Figure 6.16) demonstrates the basic structure of the bridge. Up to this step in the refinement, only ten rules are necessary. The lower rows of Figure 6.16 show how three different variations for the passageway under the bridge adapt for different slopes of the bridge. Note that the fourth row inserts arches into the non-convex geometry surrounding the big arch, which takes guidance from their scope, which is not set to the convex hull after splitting the non-convex surroundings of the main passageway. For these variations of the bridge the width w and the length l have been kept constant.

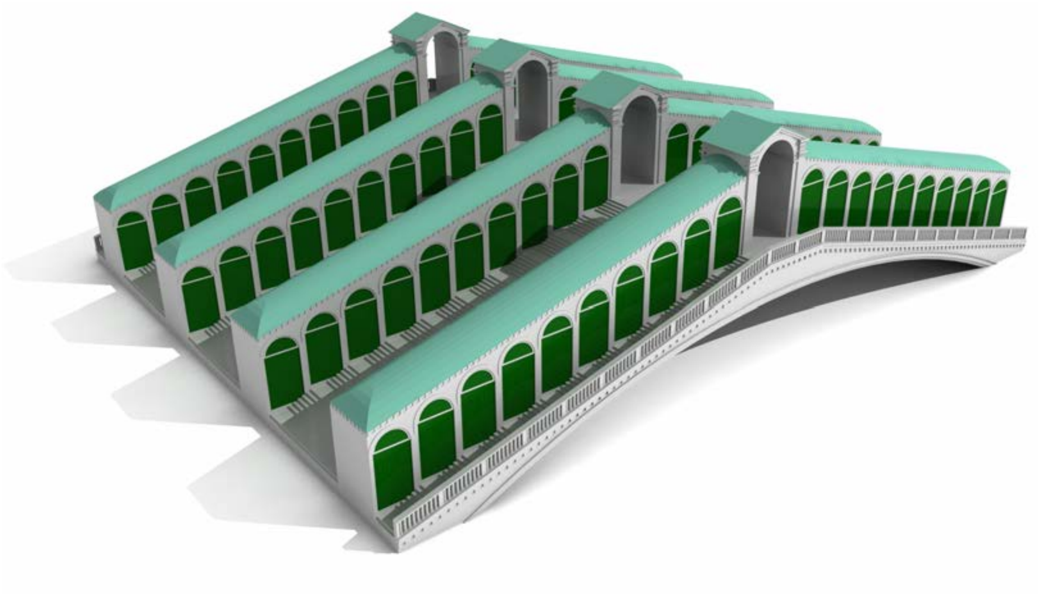


Fig. 6.15 Procedural model of the Rialto Bridge that adapts to changes in width with more or less lines of market stalls and to changes in length with more or less individual shops by keeping the size of the gates in the middle the same.

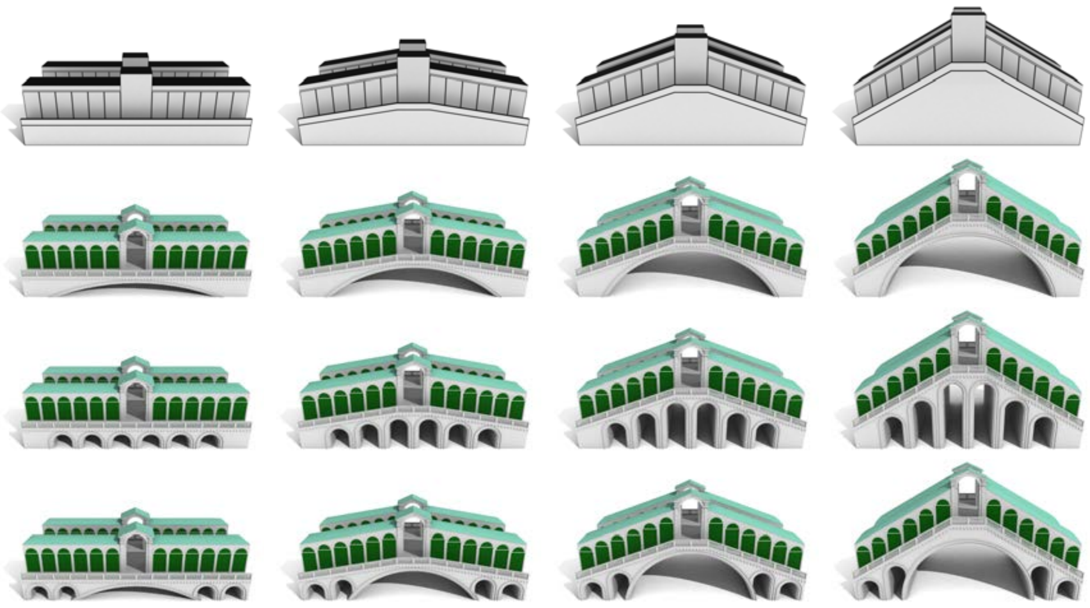


Fig. 6.16 Variations of the bridge example. The slope of the bridge varies from zero degrees (left column) in ten degree steps to thirty degrees (right column). The height of the bridge changes accordingly. The first row shows an intermediate state of the subdivision process. The convex shapes, in which arches and other details are inserted to, can be seen at this stage. The remaining three rows show different designs for the passageway beneath the bridge. Note that all arches adapt to their scopes in a procedural manner. The arches are not scaled or sheared to fit the scopes. (image source: Thaller et al. [TKZ*13b])

The input shape for the bridge is a box of appropriate dimensions; parameters for the main Bridge rule are the slope's angle α and the height h of the bridge. The ten rules to describe the basic structure of the bridge begin with the following:

Bridge(α, h) $\rightsquigarrow$ **Subdivide**(**n**=**u**=X, (A,B)=**extremes**(X), 1r, *mid-wdt*, 1r)
{Side1(α, h), Mid1(h), **mirror**(X) Side1(α, h)}
Side1(α, h) $\rightsquigarrow$ **Subdivide**(**n**=($-\sin\alpha, 0, \cos\alpha$), **u**=Z,
(A,B)=**extremes**(Z−X), h, *arcade-hgt*, 1r)
{Below, Side2, **void**}
Mid1(h) $\rightsquigarrow$ **Subdivide**(**n**=**u**=Z, (A,B)=**extremes**(Z), h, *arcade-hgt*, 1r)
{Below, Mid2, **void**}
Below∗ $\rightsquigarrow$ **Merge**()
{Passageway}

The Side1 rule introduces the slant of the bridge. It uses the diagonal direction Z-X to pick the limit points A and B. The point A is therefore the point nearest the center of the bridge on the bottom of the Side1 shape. The height h is measured upwards ($\vec{u} = Z$) from there.

After execution of Mid1 and Side1, the overall shape of the bridge is set. The later rules do not refer to the parameters α and h anymore; rather, they take their guidance from the scopes themselves. The Below∗ rule is a non-context-free rule that collects all shapes with label Below and merges them into a single shape.

The structure of the bridge is completed with the following rules:

Side2 $\rightsquigarrow$ **Subdivide**(**n**=**u**=Y, (A,B)=**extremes**(Y),
outer-wdt, *arcade-wdt*, 1r, *arcade-wdt*, *outer-wdt*)
{Outer1, Arcade1, Stairs, **mirror**(Y) Arcade1, **mirror**(Y) Outer1}
Mid2 $\rightsquigarrow$ **Subdivide**(**n**=**u**=Y, (A,B)=**extremes**(Y),
outer-wdt, *arcade-wdt*, 1r, *arcade-wdt*, *outer-wdt*)
{Outer1, Gate, **void**, **mirror**(Y) Gate, **mirror**(Y) Outer1}
Outer1 $\rightsquigarrow$ **Subdivide**(**n**=**u**=Y, (A,B)=**extremes**(Y), *balustrade-wdt*, 1r)
{Outer2, Stairs}
Outer2 $\rightsquigarrow$ **Subdivide**(**n**=**raycast**(−Z), **u**=Z, (A,B)=**extremes**(Z), *balustrade-hgt*, 1r)
{Balustrade, **void**}
Arcade1 $\rightsquigarrow$ **Subdivide**(**n**=**raycast**(−Z), **u**=Z, (A,B)=**extremes**(Z), 1r, *deco-hgt*)
{Arcade2, DecoAndRoof}
Arcade2 $\rightsquigarrow$ **Repeat**(**n**=**u**=X, (A,B)=**extremes**(X), A∗, *arcade-element-wdt*)
{ArcadeElement}

The Mid2 and Side2 rules differ only in their use of Arcade1 and Gate as labels for the resulting shapes. For the sake of simplicity, a template rule (like the ones defined in the work of Krecklau et al. [KPK10]) has not been used to unify these two rules.

The two Outer rules generate the overall shape of the balustrade without requiring an explicit angle as a parameter. A ray cast determines the normal of the lower boundary plane, which is then used to split the shape.

To complete the model of the bridge further rules that are not listed here are necessary. These rules will refine the remaining non-terminal shapes to achieve more detail. These rules continue in the following ways:

- The ArcadeElement rule continues by inscribing an arch to the sheared box, which describes the corresponding shape. To create the borders of the arch frame-split operations in combination with subdivide operations are applied.

- The DecoAndRoof rule creates the roof for the market stalls through an extrude operation and the decorative line that runs above the market stalls and below the roof. The round roof is realized through application of arch operations. The creation of the roof is detailed in Figure 6.17.
- The Balustrade rule creates the balustrade through application of a repeat operation for the individual balustrade segments and the pillars within them. There is one segment for each market stall and two segments for the gate. Each segment consists of nine pillars. The balustrade decoration is achieved through sequential application of extrude operations. Figure 6.18 illustrates the separate extrusion steps.
- The Gate rule trims its corresponding shape through two planes introducing the triangle-shaped roof, which is refined by application of frame-split and extrude operations. The top parts of the pillars are refined through extrusions and the arch is inscribed in the convex shape, which is bounded by the pillars and the roof. The border of the arch is again realized through a frame-split operation.
- The Passageway rule fits a segmental arch in the convex shape defined by the slope and the top part of the bridge. The decoration of this arch is achieved through sequential application of frame-split and extrude operations.
- The Stair rule is executed at two places: in the Side2 rule for the middle stairs and in the Outer1 rule for the stairs in between the shops and the balustrade. Stairs are realized through a repeat operation. Based on the slope the size of the even spaces that are regularly placed in the stairs is influenced. At steep slopes these even spaces vanish and at low slopes no stairs are inserted at all.

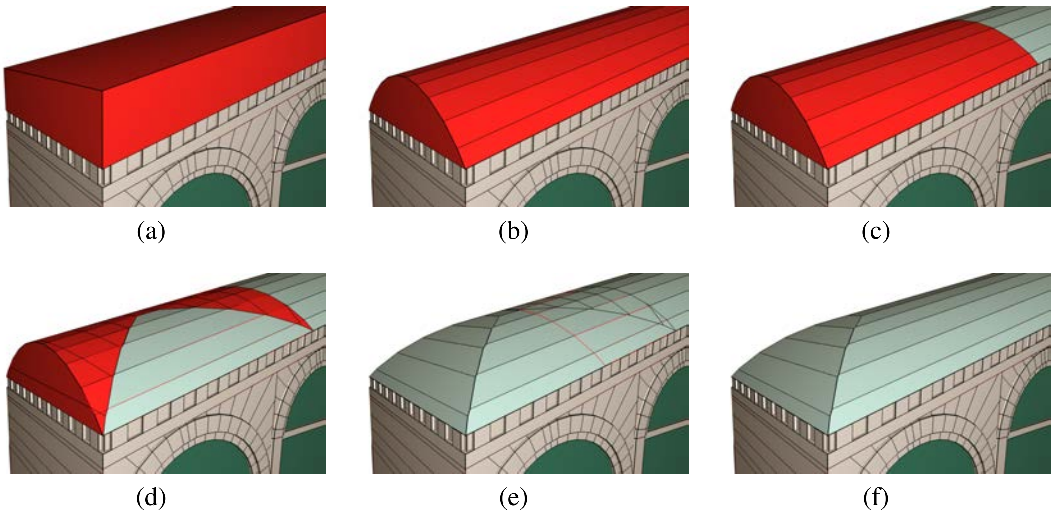


Fig. 6.17 Creation of the roof for the market stalls. Final parts are colored in the greenish roof color and parts that will be refined further are colored red. First the space for the roof is extruded (a). Afterwards an arch operation is applied in the slanted direction of the roof (b). To achieve a rounded ending the affected part is split off (c) and an arch operation is applied along the normal of the plane spanned by the up-direction and the slanted direction of the roof. (d). The non-convex part of this arch operation is split in half and the outer part is discarded (e). To finalize the shape of the roof the convex hull of all separate parts is calculated (f).

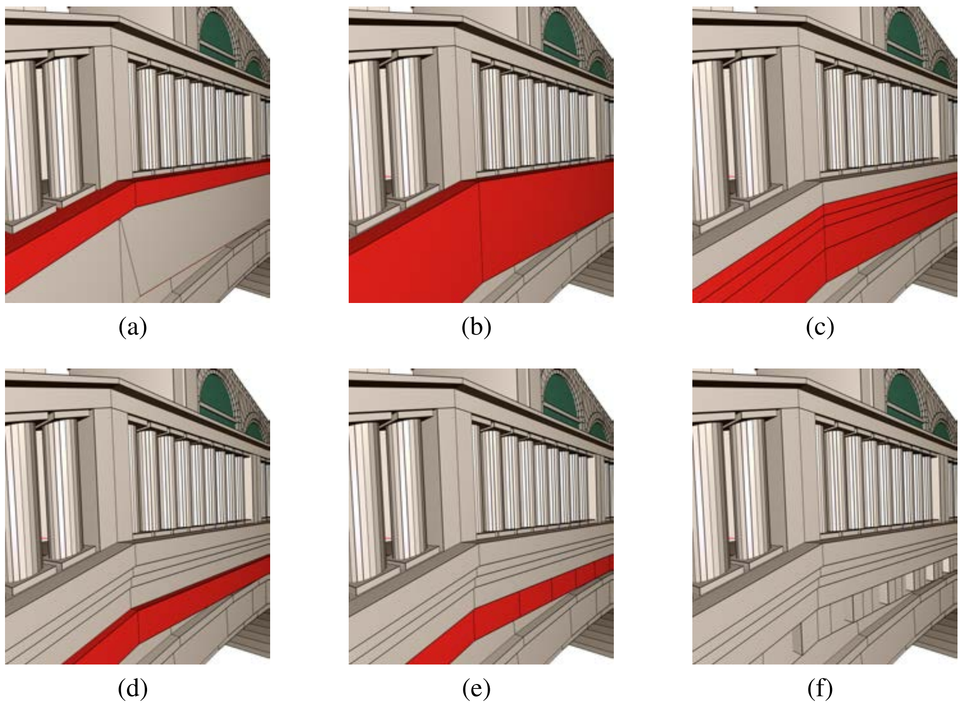


Fig. 6.18 The lower balustrade decoration is created through a sequence of extrusions. Final parts are colored grey and parts that will be refined further are colored red. The lower part of the balustrade is first extruded forward (a) and the extruded part is then extruded downward (b). Next, the different parts of the profile are added (c) and split off to achieve steps of different length (d). Finally a `repeat` operation is used to partition the lowest part into sections(e), which are refined to obtain the final result (f).

6.2.3 A Procedural Wall Model inspired by the Great Wall of China

Deformation-aware split grammar methods (see Section 4.3) allow to specify objects with structural regularity that adapt to deformations in space, like the Great Wall of China, which is a massive building that is even visible from outer space. This wall that spans mountain ranges and overcomes several elevation differences inspired the example featured in this section to demonstrate the application of straight splits in deformation-aware split grammars. This is no attempt to reconstruct the whole wall accurately; the terrain and wall placement is done manually and there is no relation to the real thing. Figure 6.19 shows the result of this reconstruction, which I did in cooperation with Edelsbrunner in our work [ZTK*14]. Wall segments are deformed to connect two towers. The repetition of elements adapts to the space provided by the deformation as seen in the crenelation and bricks of the wall. Measurements are taken along the deformation to calculate the number of objects to place. Straight splits are introduced in the deformed wall for realization of level stairs and bricks. This section details the steps necessary to achieve this result.

Generation of the Basic Wall Model. The highly regular structure of a large stonewall that stretches across a landscape makes it an ideal candidate for grammatical representation. Its an especially fitting example for deformation-aware split grammars because its path might follow a curve and should adapt to elevation differences accordingly.

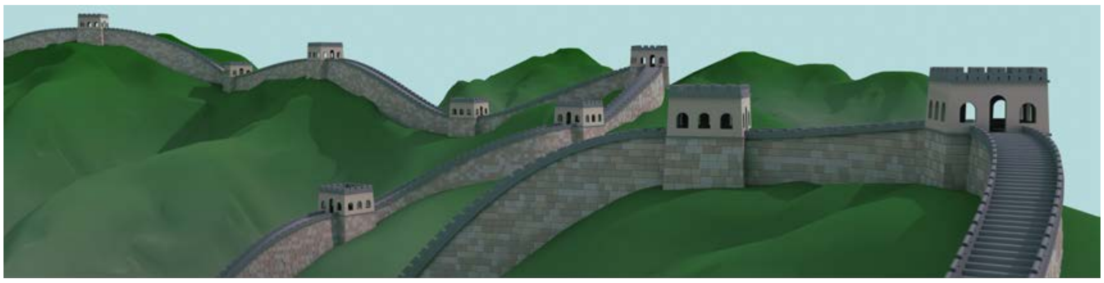


Fig. 6.19 Wall model running over a terrain inspired by the Great Wall of China. The repetition of elements adapts to the space provided by the deformation process as seen in the crenelation and bricks of the wall. Measurements are taken along the deformation to calculate the number of objects to place. Straight splits are introduced in the deformed wall for realization of level stairs and bricks. (image source: Zmugg et al. [ZTK*14])

The basic model consists of two parts: towers and wall segments. Both are realized through a split grammar applied to a box shape. For a wall segment, the shape is divided into the outer parts for the crenelation and bricks and the inner part for the walkway. The crenelation and the bricks are realized through a deformation-aware `repeat` operation with the effect that the amount of bricks and merlons adapt to the applied deformation. The size and color of the bricks are furthermore randomly generated to ensure that each piece has its unique look. The models for the towers are slightly tapered and easily generated through the use of convex polyhedra as shape representation. The same rule for bricks and merlons are also inserted here, but – except for color and size of the bricks – the amount stays the same for each tower because they will not be deformed.

The deformation of each wall segment is dependent on the location of the two towers that will be connected and is realized through three separate deformations steps that are illustrated on a simple shape in Figure 6.20. The first deformation in Figure 6.20(b) step takes the lower part of the entire wall segment (without the crenelation part) and introduces a steep, but not vertical, slope on the side of the wall. The following two deformations are used to connect the two spatially different located towers. The first deformation in Figure 6.20(c) is used to regulate the height difference and the second one in Figure 6.20(d) is used to overcome the difference on the horizontal plane.

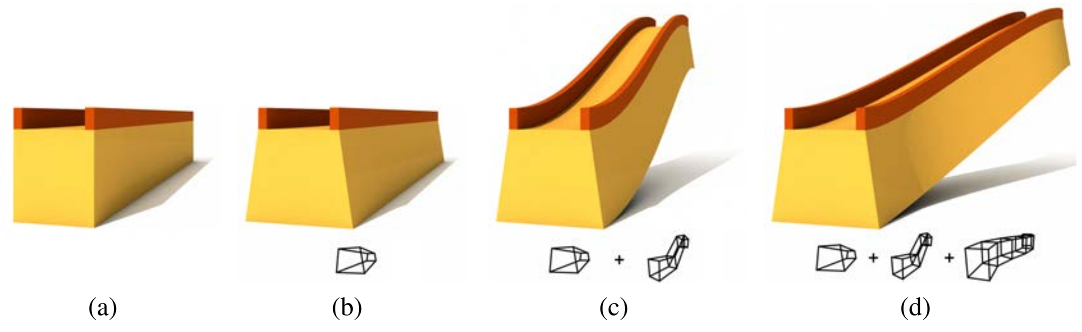


Fig. 6.20 Illustration of a nested application of free-form deformations on a wall to connect two towers with different locations. Starting from a simple wall shape (a) three steps of deformations are applied. First only the base (yellow) is affected by a widening deformation (b). For the wall to serve as a connector between two towers, afterwards vertical (c) and horizontal (d) deformations are applied. (image source: Zmugg et al. [ZTK*14])

This basic model of a wall, which connects two towers $t1$ and $t2$ with individual positions and orientations, can be realized by the following split grammar rules:

```
Wall(t1,t2)           ⇝ Deform(localBB, (12,2,2),
                        calcHorizontalDeformationOffsets(t1,t2))
                        {DeformedWall1(t1,t2)}
DeformedWall1(t1,t2)  ⇝ Deform(localBB, (4,2,2),
                        calcVerticalDeformationOffsets(t1,t2))
                        {DeformedWall2}
DeformedWall2         ⇝ Subdivide(Z, 1r, crenelationH)
                        {Base, Top}
Base                  ⇝ Deform(localBB,(2,2,2), calcBaseDeformationOffsets())
                        {DeformedBase}
DeformedBase          ⇝ Subdivide(Y, brickW, 1r, brickW)
                        {BrickWall, Walkway, mirror(Y) BrickWall}
BrickWall             ⇝ Repeat(Z, A∗, brickH)
                        {BrickRow}
BrickRow              ⇝ RandomRepeatD(X, A∗ brickW, rayIntersect(Y))
                        {Brick}
Top                   ⇝ Subdivide(Y, brickW, 1r, brickW)
                        {Crenelation, void, mirror(Y) Crenelation}
Crenelation           ⇝ RepeatD(X, A(BA)∗, B, merlonW, spaceW, rayIntersect(Y))
                        {Merlon, MerlonSpace}
```

Rules are not provided for all non-terminal shape labels that appear. All labels without an associated rule insert further detail like the shape of the single bricks or merlons. To keep the code as simple and concise as possible these replacements steps have been skipped in this listing.

Note that, as with the geometry, deformations are also specified in a coarse to fine manner. This means for the deformations in Figure 6.20 that the horizontal deformation in Figure 6.20(d) is specified first in the Wall rule, the vertical deformation in Figure 6.20(c) right after in the DeformedWall1 rule, and the smaller base deformation in Figure 6.20(b) is specified later in the hierarchy in the Base rule. Furthermore, note that only the first two deformations are dependent on the location and orientation of the two towers.

Of additional interest is the randomized repeatD operation utilized in the BrickRow rule. This rule takes a minimum size and then partitions the shape in pieces of at least that size. The sizes of the pieces are chosen randomly; in case of the bricks this is done to enhance the impression of the wall.

Introduction of Straight Splits in the Wall Model. Two problems arise within the basic model, which was been described so far. First, all splits done to realize different levels of bricks will follow the course of the deformation. This leads to deformed bricks, which are whether neither practical nor realistic. Second, stairs on the walkway cannot be realized in this model because the height difference is introduced by the deformation. It is not possible to introduce splits in the undeformed model that will be transformed to stairs in the deformed model. A basic wall segment with the corresponding two towers and those mentioned problems is shown in Figure 6.21(a).

The order and number of hierarchical deformation steps was in no way chosen arbitrarily. Both aforementioned problems can be solved by baking the deformed geometry at the correct spot in this deformation hierarchy. Both problems are dependent on the height difference of the wall, so by baking the geometry of the lower wall after the height deformation step (see

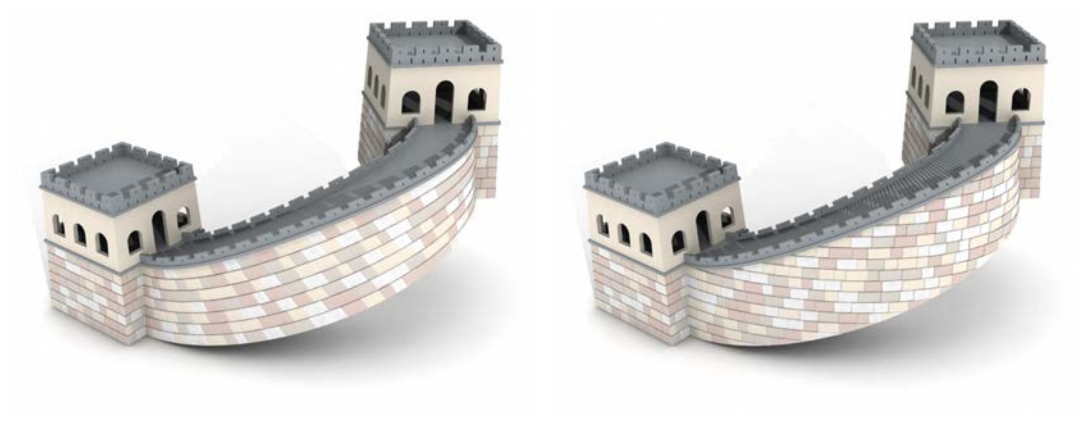


Fig. 6.21 Without baking the geometry it is not possible to realize level stairs and bricks in deformed wall segments (a). Split lines follow the deformations and stairs cannot be realized due to the fact that the height offset is introduced by a deformation and is not accessible in the original undeformed model. However, by baking the geometry after the height deformation these level structures can be introduced easily (b). The amount of stairs and bricks still adapt to all deformations that are applied afterwards. (image adapted from Zmugg et al. [ZTK*14])

Figure 4.24(c)) straight splits (through a repeatD operation) can be used to split level stairs and bricks, which adapt to the remaining deformations, into the wall. It is important that this baking step and the splits are done before the final deformation step. This final deformation step introduces the stretching of the wall and therefore, even though the geometry was baked before, the amount of stairs and bricks can adapt to this deformation. Note that this is not necessary for the crenelation part, which is not included in the bake operation. The crenelation should follow and adapt to all deformations. The final model with level stairs and bricks in shown in Figure 6.21(b).

The updated split grammar is as follows:

Wall(*t1*,*t2*)	⇝	**Deform**(localBB, (12,2,2), **calcHorizontalDeformationOffsets**(*t1*,*t2*)) {DeformedWall1(*t1*,*t2*)}
DeformedWall1(*t1*,*t2*)	⇝	**Deform**(localBB, (4,2,2), **calcVerticalDeformationOffsets**(*t1*,*t2*)) {DeformedWall2}
DeformedWall2	⇝	**Subdivide**(Z, 1r, *crenelationH*) {Base, Top}
Base	⇝	**Deform**(localBB,(2,2,2), **calcBaseDeformationOffsets**()) {DeformedBase}
DeformedBase	⇝	**Subdivide**(Y, *brickW*, 1r, *brickW*) {BrickWall, Walkway, **mirror**(Y) BrickWall}
BrickWall	⇝	**Bake**(2) {BrickWall1}
BrickWall1	⇝	**Repeat**(Z, A∗, *brickH*) {BrickRow}
BrickRow	⇝	**RandomRepeatD**(X, *brickW*, **rayIntersect**(Y)) {Brick}
Walkway	⇝	**Bake**(2) {Stairs}

Stairs	⇝	**RepeatD**(X, A∗, *stairW*, **rayIntersect**(Z)) {Step}
Top	⇝	**Subdivide**(Y, *brickW*, 1r, *brickW*) {Crenelation, **void**, **mirror**(Y) Crenelation}
Crenelation	⇝	**RepeatD**(X, A(BA)∗, B, *merlonW*, *spaceW*, **rayIntersect**(Y)) {Merlon, MerlonSpace}

Changes are introduced at two locations. First the BrickWall rule has been modified. A bake operation is applied before the wall is split in z-direction to introduce the individual brick rows. This operation takes the innermost two deformations – the base deformation and the vertical deformation – and applies them to the shapes labeled with the name BrickWall. Through this the repeat operation in the BrickWall1 rule can introduce straight splits to the deformed wall. The randomized repeatD operation in BrickRow rule still adapts to the horizontal deformation because this deformation has not been baked. The second change is the addition of stairs. The Walkway rule also bakes the innermost two deformation to enable the following Stairs rule to use a repeatD operation – that adapts to the remaining deformation – to split the before unused Walkway part into single steps.

6.2.4 A Procedural Model of the Eiffel Tower

This section highlights the work done by Eger in context of his Master's thesis [Ege13] with the title *"Locally Context-Sensitive Shape Grammars"*. He developed a variant of a split grammar on convex polyhedra and generated a procedural model of the Eiffel Tower. Through developing a formalism that allows intensive use of interconnected structures, aligned struttings throughout different levels of the tower, the merging of the different pillars, and connecting parts for different beams have been realized. The model, furthermore, features different levels of detail from a sparse wire frame model to a highly detailed model that even covers the single rivets. For more detail on how this model was realized see the associated Master's thesis.

A rendering of a model with a high level of detail (without rivets) together with zoomed in detail views is shown in Figure 6.22. Different level of detail with no connecting beams or less detailed beams are shown in Figure 6.23(a). Since this is a procedural model a change of parameters leads to variations of the original model. Variations with different height proportions and angles are shown in Figure 6.23(b).

6.3 Procedural Generation of Buildings

Content creation for the movie and entertainment industry motivated procedural generation of entire cities. Dependent on the application, buildings are realized within these cities with more or less detail. These levels of detail can range from simple "paper" models for example for flight simulations, to buildings with highly detailed façades and interiors for applications like first-person shooter games.

A building is a collection of different interconnecting design elements, which all deserve their own analysis, like the one that has been done on windows in Section 5.2. The interiors of buildings including room arrangements with connecting structures like doors and staircases are an especially hard and prominent topic. This section shows the results achieved through split grammars in the domain of procedural generation of buildings. Aside from windows,

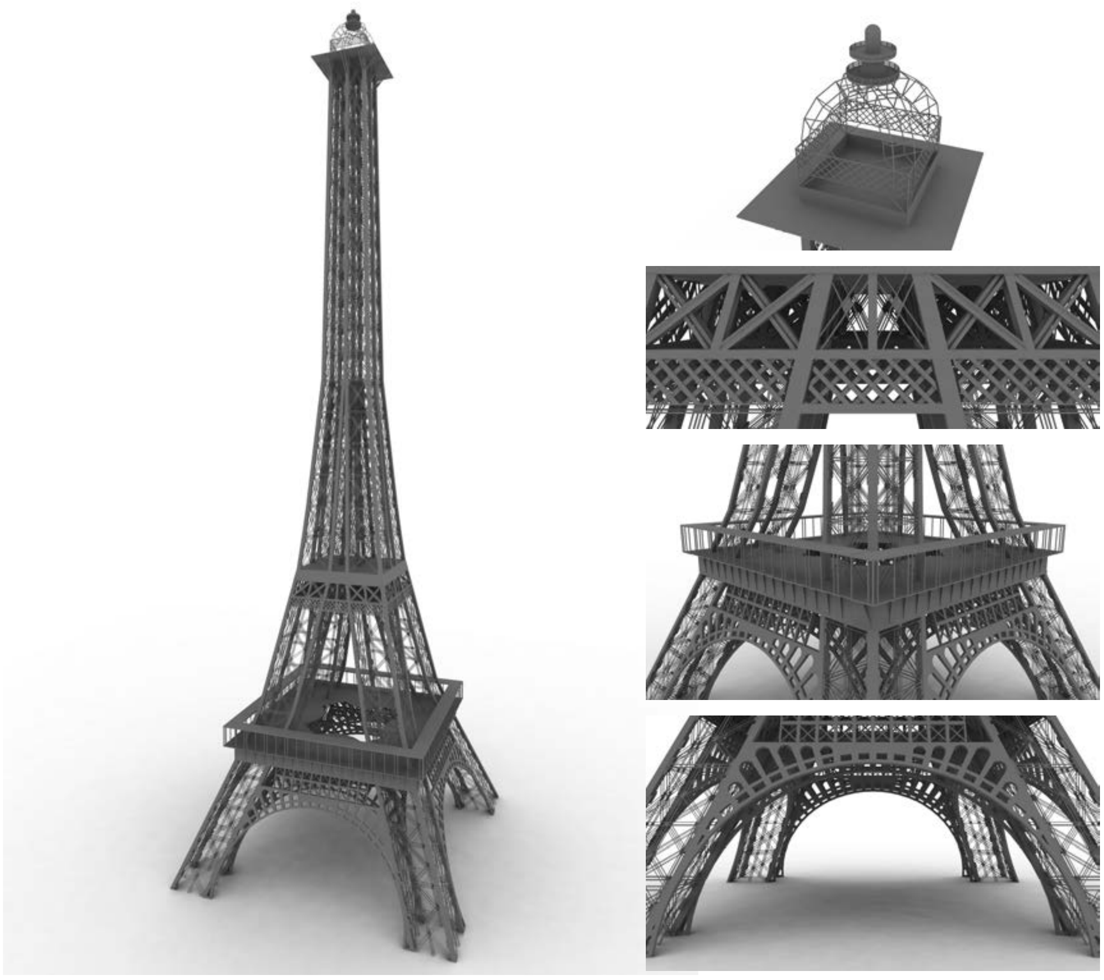


Fig. 6.22 A procedural model of the Eiffel Tower in Paris realized through split grammars based on convex polyhedra within the GML. Zoomed in detail renderings of important parts are shown on the right. (image source: Eger [Ege13])

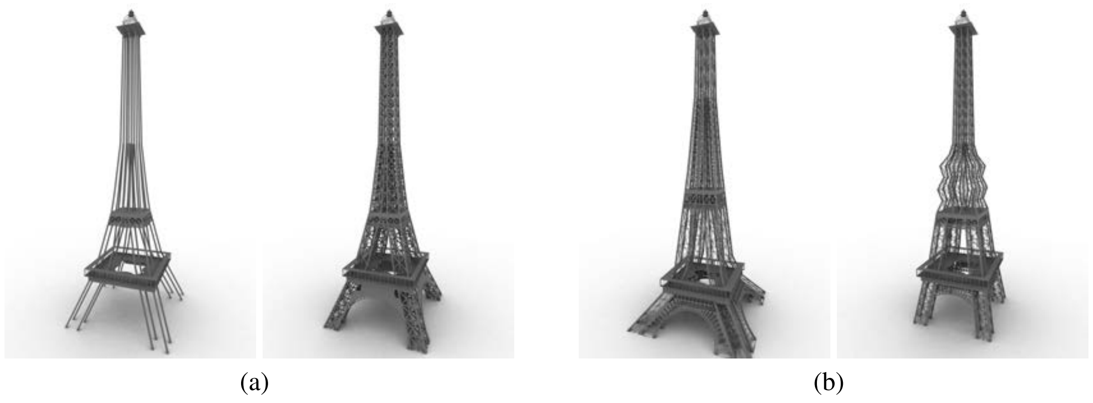


Fig. 6.23 Different detail levels of the Eiffel Tower model (a) as well as models reached through variations of certain parameters like height proportions and angles (a). (image source: Eger [Ege13])

staircases (see Section 6.3.1), office buildings (see Section 6.3.2), and deformed and cartoonish architecture (see Section 6.3.3) have been explored by me.

6.3.1 Procedural Staircases

When realizing a procedural model of a building with interior staircases are an integral part because they are necessary for accessing different floors of a building. Dependent on the available space, staircases need to be realized in different shaped volumes.

To showcase the power of split grammars on convex polyhedra (see Section 4.2) I realized staircases in various shapes. For realizing staircases in arbitrary, but reasonable, surroundings step shapes do not necessarily have to stay the same. As within the example of the winding staircase in Section 4.2.2 the scope for each step may be different, but convex. It is desirable to provide only one rule for the steps instead of many different rules or geometries dealing with different situations. This highlights the advantages of split grammars on convex polyhedra. As explained in detail in Section 4.2.1, query operations are utilized to find important features of the scope, which are again used to determine the shape of the step.

To show the versatility of this Step rule, staircases have been constructed in various extreme shapes. These non-convex shapes have been generated from several poly-lines forming letters. For each of these shapes, one or more 'walking lines' – the lines along which all stairs have the same depth – had been created manually. This line was then used to split the shape into the individual step shapes with the help of the polyline-split. Finally, the same Step rule was applied to all of these shapes. The result of this is shown in Figure 6.24 and 6.25.

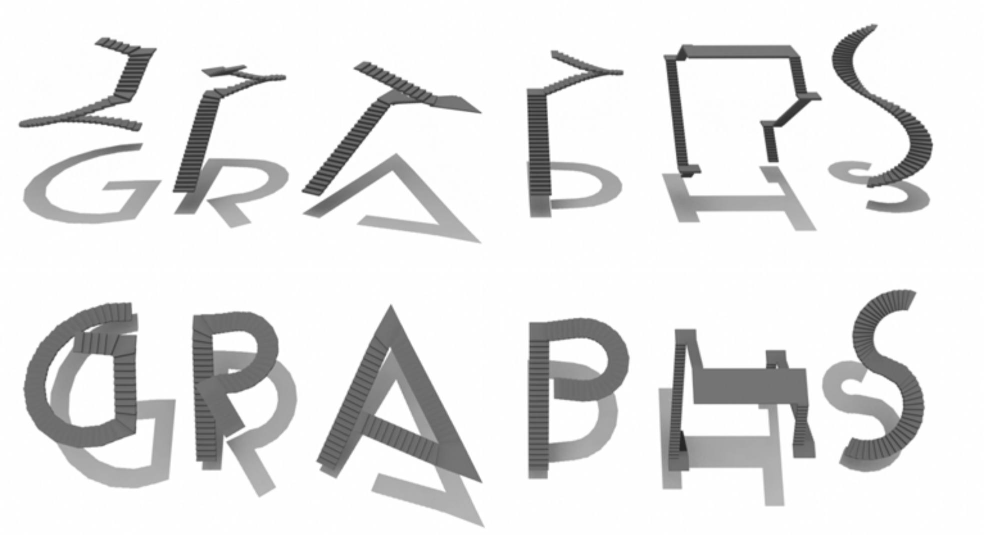


Fig. 6.24 Stairs inserted into various non-convex geometries forming letters that read "GRAPHS". For all of these letter-like shapes, except for the 'H', one walking line is sufficient to create the stairs.

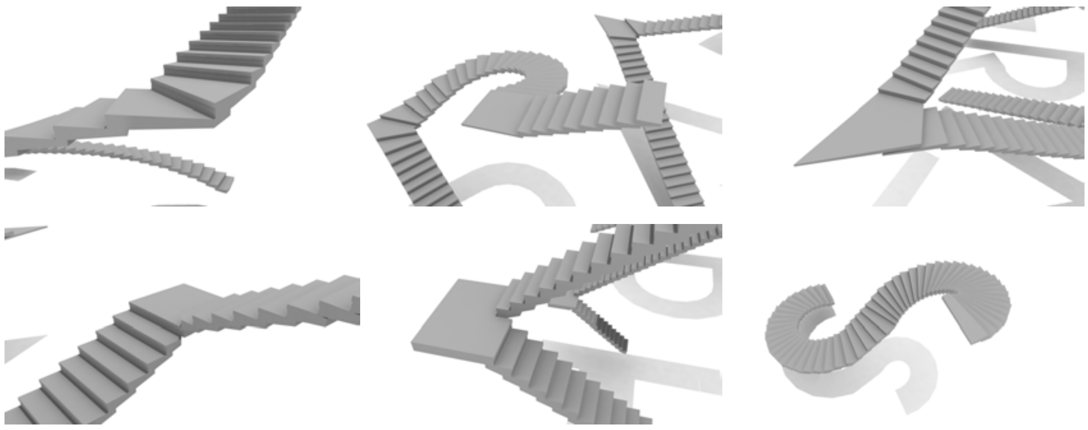


Fig. 6.25 Closeup views of the step shapes of the staircases shown in Figure 6.24. The shape of the steps is automatically generated in the correct way based on the available convex scope.

6.3.2 Procedural Office Buildings

Oblong office buildings have a very simple internal structure which allows to demonstrate the volumetric approach of the frame-split operation, which has been introduced in context of split grammars on convex polyhedra (see Section 4.2). This section includes extended explanations from examples that have been realized in context of our work [TKZ*13b, ZTK*13, ZTK*14].

Defining the Structure of Office Buildings. The inner part of the frame-split is used in the OfficeBuilding rule to design the interior of a simple structured office building (see Figure 6.26). Each level of the split was used for a different part of the building such as the façade, rooms, hallway, and courtyard wall. Attributes that are assigned to the scope's bounding planes define the way how the parts of the frame-split operation, which originate from these bounding planes, are processed further. All parts of the sequential application of the frame-split operation are influenced by these attributes, for example, the side with an entrance has a door in the façade and this door has a connection to the hallway, which further leads to a staircase.

The OfficeBuilding rule can be applied to any reasonable convex ground plan. Figure 6.26(c) and (d) show the result of applying this OfficeBuilding rule to several ground plans generated through a manually constructed street layout.

The interior that has been created through split grammars for these buildings is still pretty simple. Realization of complex room arrangements involving non-convex rooms through split grammars is still an open problem.

Alignment of Windows and Interior Walls. Façades and interior walls both follow rules that can be specified by split grammars. The layout of the façade rarely depends on the placement of interior walls, but the placement of the interior walls need to respect the structures implied by the façade. Thus, the rules for specifying the façade layout should be created independently from the rules for the interior.

With the help of convex polyhedra and interconnected structures, variants of the subdivide and repeat operations can be defined that shift the split planes up to a specified maximum distance to avoid intersections between the walls and a list of shapes that is given as an additional parameter – the so-called avoidance volumes. These operations, which have been developed

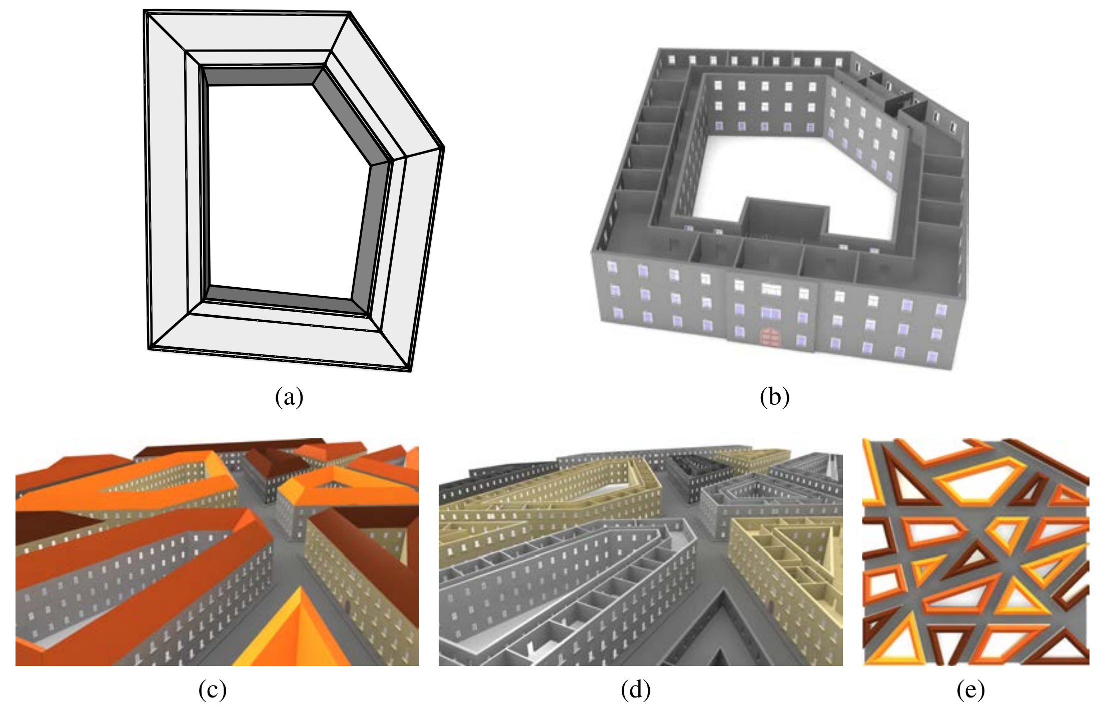


Fig. 6.26 Application of the OfficeBuilding rule. The frame-split operation is used sequentially to create the interior of a simple structured office building. Each level of the split was used for a different part of the building (a) such as the façade, rooms, hallway, and courtyard wall. These parts can then be further refined based on initial attributes. For example, the side with an entrance has a door in the façade and this door has a connection to the hallway, which further leads to a staircase. The OfficeBuilding rule can be applied to any reasonable convex ground plan ((c) and (d)). A manually constructed street layout (e) was used to generate a set of ground plans. (image adapted from Thaller et al. [TKZ*13b])

by Thaller and me, can be considered as complementary to the snap lines and occlusion query mechanisms of CGA Shape [MWH*06].

In Figure 6.27 these operations are applied in the office building example. The avoidance volumes are created by extruding the scopes of the windows inwards (orange). These shapes are collected and passed to the grammar evaluation tree for the interior, which then respects these volumes. Thus, each window locally defines a volume that walls, which are defined elsewhere, will avoid in their placement.

Implementation of City Blocks. Separate houses in one block can also be realized through splitting the initial building shape into several parts. Attributes define that the planes in between the neighboring houses are not included in the frame-split operation. Each part of the building is a separate application of the OfficeBuilding rule and can then be generated with different parameters. An example for this is shown in Figure 6.28(a). Structures that span more than one floor, such as staircases, have been also realized in this example (see Figure 6.28(b) and (c)).

Curved Office Buildings. As buildings do not necessarily have to feature straight façades when they follow the course of a street, office building have also been realized through deformation-aware split grammars as presented in Section 4.3.

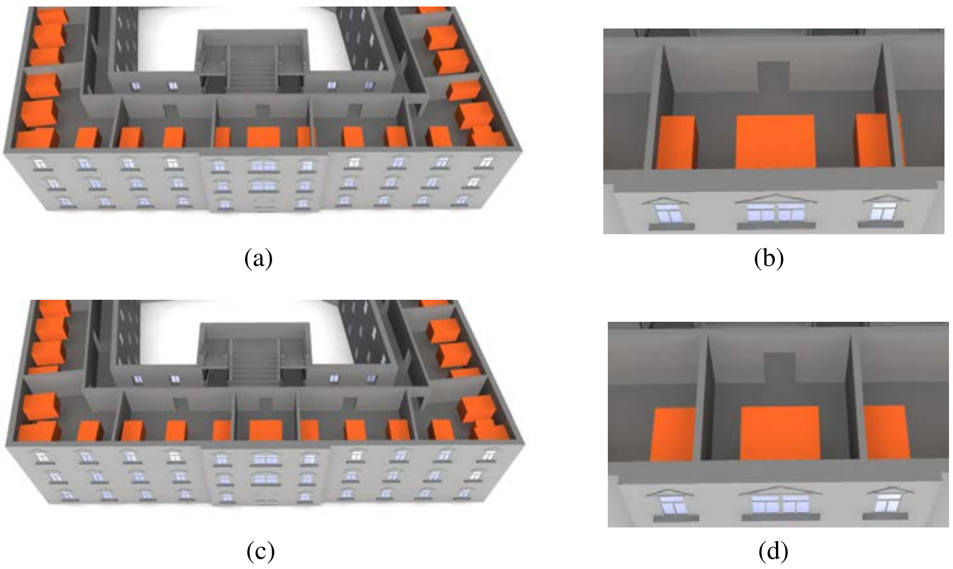


Fig. 6.27 If a normal repeat operation is used to place interior walls independently of the façade, they will sometimes violate the forbidden space (orange) in front of a window ((a) and (b)). Using the avoidance volumes mechanism, the walls are shifted sideways from their ideal (equally-spaced) positions until they no longer intersect a forbidden space ((c) and (d)). (image source: Thaller et al. [TKZ*13b])

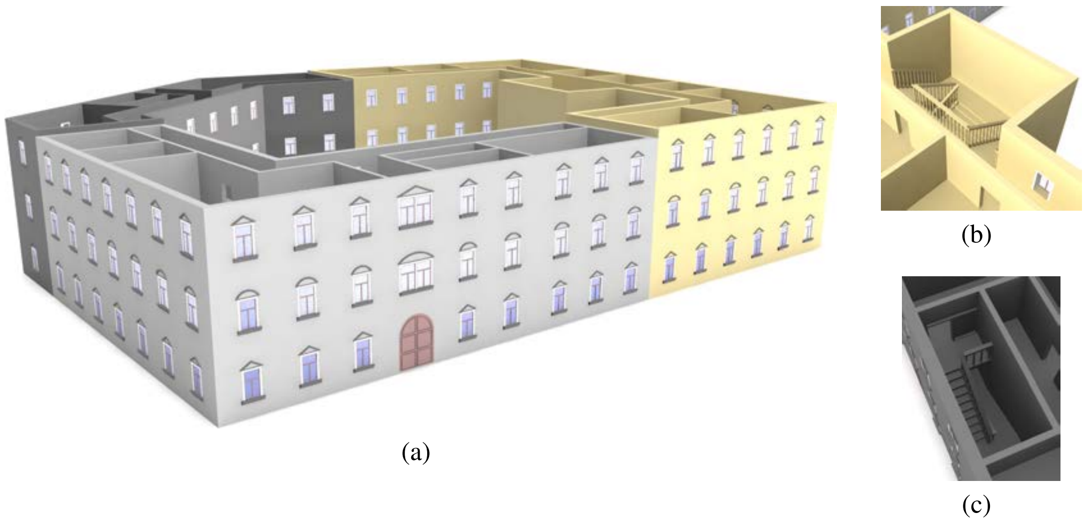
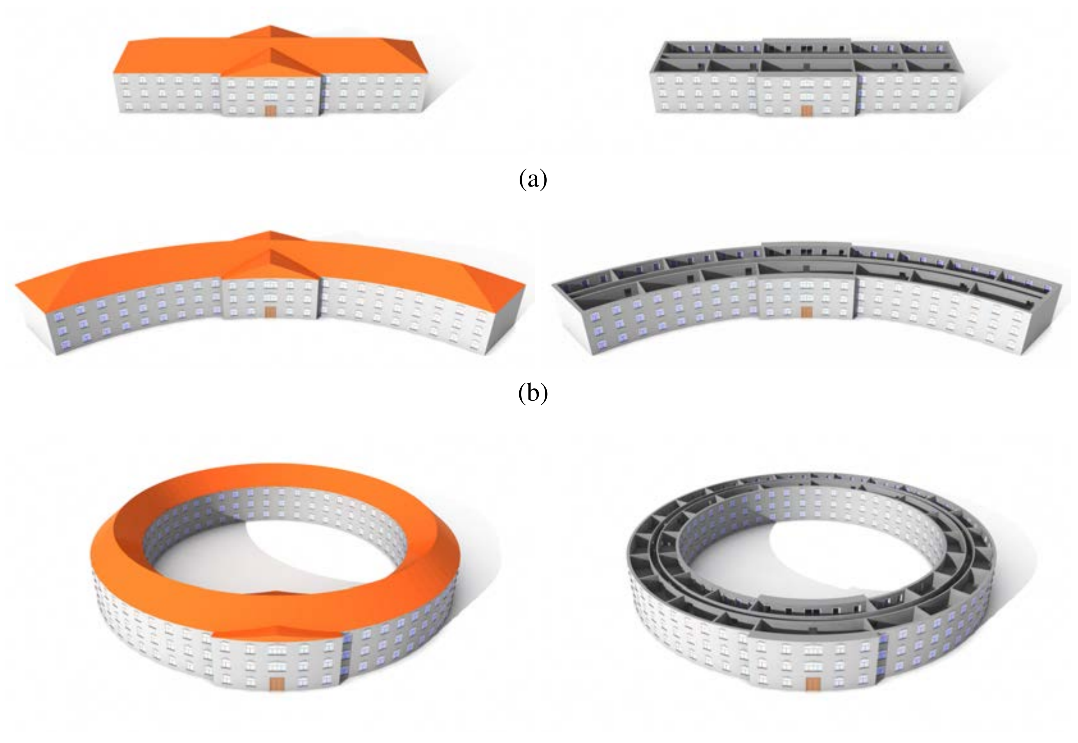


Fig. 6.28 A city block (a) is realized using multiple execution of the OfficeBuilding rule. The initial geometry is split into parts and the house generating rule is applied to each part separately with different parameters (for example the height). Attributes define which sides of the houses have an entrance. Staircases that span over several floors have been realized as well ((b) and (c)). (image adapted from Thaller et al. [TKZ*13b])


(a)

(b)

(c)

Fig. 6.29 A building with room layout, defined using a split grammar approach (a), adapts accordingly under different deformations that approximate circles and circle segments. For these deformations only the deformation on the course structure of the building has been added to the grammar, the number of rooms and windows is handled automatically with the repeatD operation on deformed scopes. The circle segment deformation (b) takes the building as shown in (a). For topological changes (c) the grammar rules for the roof and the left and right boundary walls of (a) have been adapted and an appropriate deformation has been applied to achieve a C^1-continuous transition. (image source: Zmugg et al. [ZTK*13])

The example features an oblong building which is bent into different shapes using deformations (see Figure 6.29) that approximate circular shapes. The building structure is again described through the frame-split operation and is split into hallway and office parts, and automatic positioning of rooms using a repeatD rule. Similarly, windows on façades are spaced using repeatD. Therefore, the window and room placement adapts automatically under deformations. The number of windows and rooms changes accordingly, although only one deformation rule has been inserted in an early replacement step that corresponds to the coarse building layout.

To achieve topological changes, like in the circular building in Figure 6.29(c), the initial split grammar has to be adapted on the roof and the left and right boundary walls to guarantee a smooth transition. Furthermore, to achieve a C^1-continuous transition at the connection points, an appropriate deformation is necessary.

6.3.3 Procedural Deformed and Cartoonish Architecture

Deformation-aware Split Grammars (see Section 4.3) have been designed with the desire in mind to create architecture that does not necessarily need to feature only straight planar parts. This section details the generation of several use cases realized with this specific addition to the split grammar formalism. If not stated otherwise, the examples presented in this section have been implemented by me in context of our work [ZTK*13, ZTK*14].

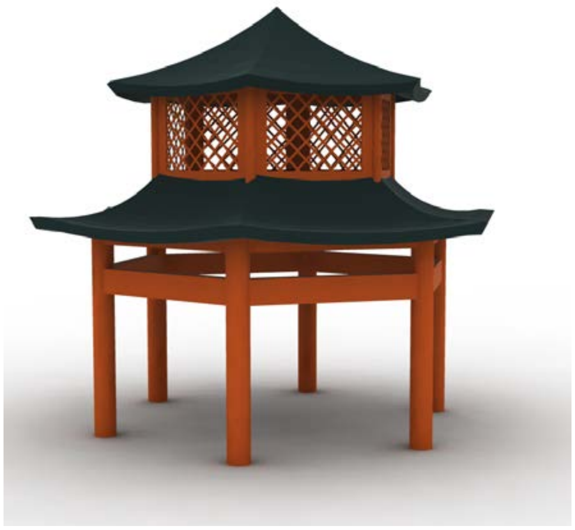
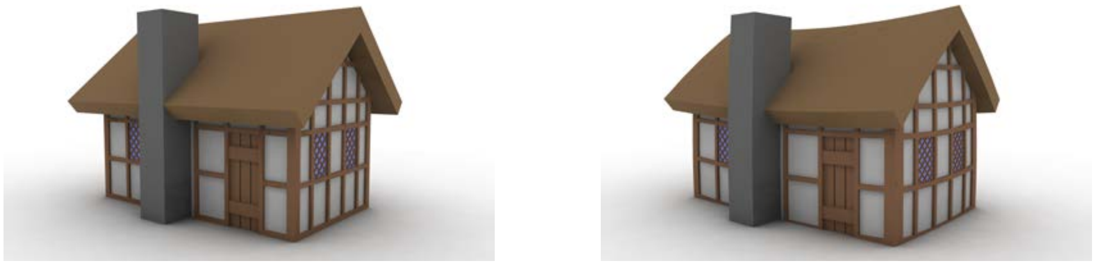
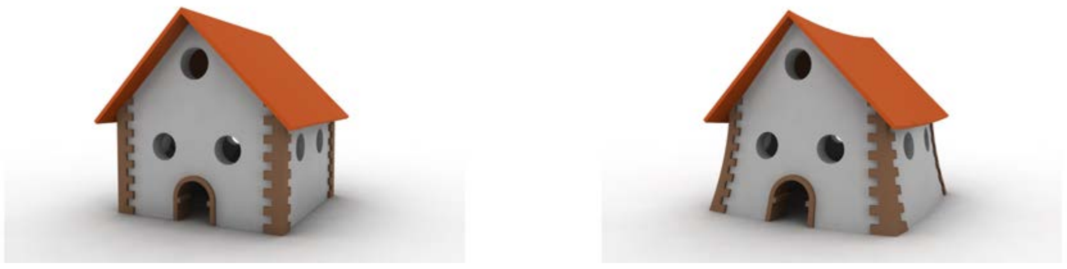


Fig. 6.30 A pagoda is realized through two layers. The upper one is placed in the hole created by the roof segments of the lower layer. The curved roof segments that are common in East Asian architecture can be approximated. (image source: Zmugg et al. [ZTK*13])

A Procedural Pagoda Model. Many elements of classical East Asian architecture (pavilions, pagodas, etc.) follow strict rules of hierarchy and symmetry and are therefore naturally suited for a grammatical representation. For the pagoda in Figure 6.30 the two stories are modeled separately. The second story is placed in the hole produced by the roof of the ground story. The main focus here lies on the curved roof parts that are approximated through deformations. Roof details, like the one realized in the work of Tenou et al. [Teo09] and Huang and Tai [HT13], cannot be applied at the current state. For this it is necessary to apply rules to results of the Boolean operations, which is not possible yet.

Procedural Medieval Architecture. Medieval buildings exhibit only slight deformations (see Figure 6.31(a)). To realize this, each wall is deformed separately, but windows and doors are not influenced by the deformations of neighboring walls. The roof as well as the timber framing gives way under the weight and is therefore slightly bent downwards to simulate the aging process. A second layer of deformations was applied to realize this aging process for walls and roofs. The chimney was declared as an exception for the Boolean operation, so it is not trimmed by the minus space of the roof.


(a)


(b)

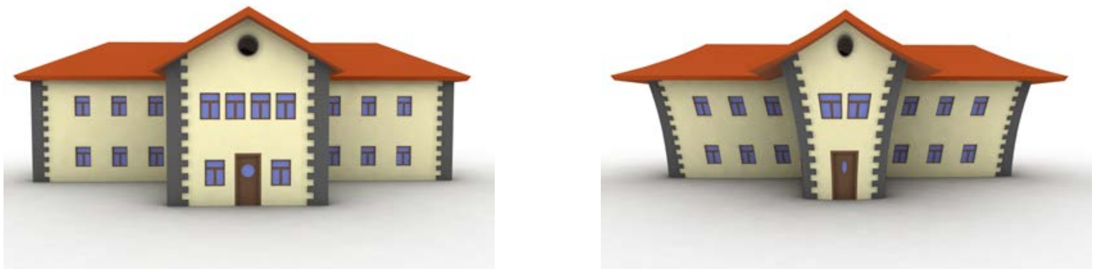

(c)

Fig. 6.31 A comparison of undeformed (left column) and deformed (right column) buildings. The medieval building (a) is only deformed slightly to simulate that the roof and the timber framing give way under the weight of the used materials. Stronger deformations are applied to the cartoonish caricatures. The simple rectangular building (b) is made bulkier through deformations, to which roof ridges and corner decorations adapt. In the third building (c) windows vanish due to the strong deformation, which reduces the space available in the separate floors. (image source: Zmugg et al. [ZTK*13])

Procedural Cartoonish Caricatures of Buildings. Cartoonish buildings can be deformed in various ways as the designer sees fit. In Figure 6.31(b) a simple rectangular building is deformed to give it a more bulky shape. More space is created, but not enough to generate more windows. The edge decoration is applied to the extended wall parts. These are combined by Boolean operations in a way that a consistent edge decoration can be guaranteed for the deformed wall parts. The roof ridge as well is a result of the deformations on both roof segments.

Another example is shown in Figure 6.31(c). The whole building is deformed with a deformation that is stronger than before. Walls are bent inwards and the separate floors are narrowed. The effect on the entrance wing of the building is that the space available in the individual floors is reduced. Therefore, all windows vanish from the ground floor and the number of windows in the second floor is diminished. As before the corner decorations are still placed accordingly and roof ridges adapt to the deformations of the corresponding roof segments.

Semantically Consistent Procedural Taverns. The tools for deformed adjacent parts together with the corresponding Boolean operations have also been integrated in the GML Compositor by me. The topic of Schwarz in his Master's seminar work [Sch14] was to design semantically consistent cartoonish taverns in this framework under my supervision. After an analysis of tavern styles that are common within fantasy-themed illustrations and computer games, he extracted a set of important parameters as well as constraints upon these parameters that fit to the characteristics of this class of buildings. This led to a semantically consistent parametric model of a tavern that has been realized within the GML Compositor. After additional refinement this model could be used in the future to create cartoonish taverns automatically on a large scale. Examples of such taverns are shown in Figure 6.32.

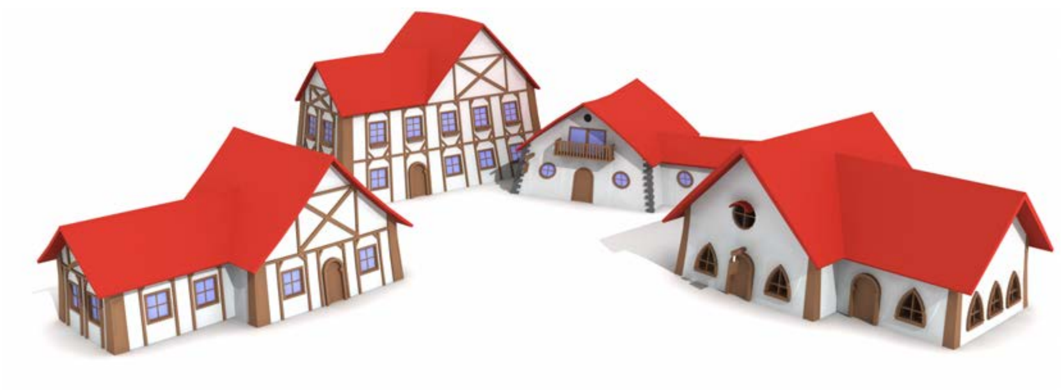


Fig. 6.32 Various kinds of taverns realized with the library of Schwarz [Sch14] inside the GML Compositor. Characteristic features of taverns in medieval video games are specific ground layouts, half-timbered constructions and door signs.

6.4 Authoring Animated Virtual Museum Exhibits

In the domain of Cultural Heritage, visualizations of digitalized artifacts can be a great asset for augmenting museum exhibits. However, single-object viewers only allow a detailed inspection of one single high quality artifact. Additional information is only brought up by separate informations points, which present further text or images. Ideally, digital exhibits feature, for example, interactive animations showing dynamically moving objects, all triggered and controlled by user interaction. Instead of showing a static object that has neither inherent story nor plot, interactive animations can be used in many different ways; they can, among other things,

- show an object in its context (e.g. excavation site),
- show how objects were utilized and employed,
- show related objects physically residing elsewhere,
- show three-dimensional comparisons with other similar objects,
- explain intricate assemblies with exploding views, or
- show transitions between different hypotheses.

If exhibits and showcases in museums are explained by displays rather than by paper notes, then these displays can also be used in other ways. Explanations can be far more illustrative when they are done using small three-dimensional animations instead by text. However, creating compelling animations is a costly endeavor even without counting the cost of high quality acquisition of digital artifacts. For the foreseeable future high-end tools like Autodesk Maya [Aut13c] will still be used for content creation and the task of game engines like Unity3D [Uni13] will be presentation purposes. With the massive increase in three-dimensional digitalization campaigns, which produce a large quantity of high quality museum artifact scans, the scalability of this established method of generating compelling animations becomes an issue. It is therefore time to research new ways to create interactive animations in a more cost and labor efficient way. This opens up a place for a new type of authoring tools that create such animations for museums.

Our work [ZTH*12] focuses on these aspects and makes a first step in the direction of streamlining the production of animated Cultural Heritage visualizations. This section is based on this work. For the task at hand, a variant of the GML Compositor (see Section 4.4) was developed. This software features two important features for the streamlining of animated visualizations of digital artifacts. First, besides the procedural modeling engine featured in the GML Compositor, a direct integration with a digital asset repository – in this case the Repository Infrastructure (RI) of 3D-COFORM [PBH*10, DTT*10] – has been integrated in the system. And second, the authoring process has been divided into three distinct stages, which are realized in different modes in the system (see Figure 6.33):

- a *designer mode* where design-oriented staff create a set of good-looking configurable scene templates containing the scene backdrop, animations, and placeholders instead of three-dimensional assets,
- a *curator mode* where one of the available scene templates is chosen and filled via drag-and-drop with high quality three-dimensional assets from the repository, and
- a *presentation mode* for viewing and presentation purposes of the generated scene.

These modes are the main contribution of this system and will be explained in detail in the following. The contributions of this work has to be attributed to two groups. Hecher and Schiffer have been responsible for the connection to the digital repository as well as the import of the models. Thaller and me have been in charge of the realization of the system within the GML Compositor. There I focused on integrating the scene graph functionality and the interactions with it. The three modes have been realized in collaboration with Thaller.

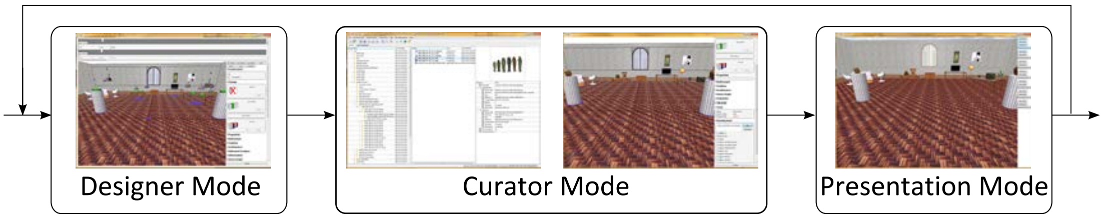


Fig. 6.33 The production of animated scenes proceeds from the designer mode to the curator mode and, finally to the presentation mode. The creation task is no linear process, but an iterative one. Changes can be made at all levels at all times due the procedural basis of the system. (image source: Zmugg et al. [ZTH*12])

6.4.1 Related Approaches for Authoring Virtual Exhibits

The augmentation of selected real exhibits through stunning three-dimensional information, such as the exploration of an Egyptian mummy at the British museum [BSly], led to an interest in three-dimensional visualizations in museums. Museums already started to digitalize all their exhibits, so usually a large pool of high quality virtual artifacts already exists.

Virtual Museums. In terms of virtual museums there are a lot of different publications. Most notably here are the work done in context of the ARCO project [WMD*04, WWWC04] and in the context of exhibition planning [MMPD08, HME*12]. In those publications similar concepts are used as in our work [ZTH*12] (such as exhibition templates), but the focus is different. While the aforementioned work focus on whole virtual exhibitions, Zmugg et al. focus at the virtual exhibits. Real museums should not be replaced, but the appreciation of augmenting physical artifacts with virtual ones should be enhanced. Therefore literature on virtual museums is only marginally related.

Single Object Viewers for Digital Exhibits. The common way to showcase virtual exhibits is using a single object viewer. One example here is the *VirtualInspector* [CPCS08, CPCS06], a viewer that can be configured with HTML pages and a bit of script code. This viewer has been used for many complementary exhibits, e.g. visitors can explore the individual chisel marks on the five meter tall David statue in Florence. In terms of efficiency through custom-tailoring a presentation software to a comprehensive collection of virtual artifacts, the *Colonia 3D* project [TSP*12] takes a rather different approach. The main focus of their work is complete historical city reconstruction. Their browser supports three interaction modes: findings mode, reconstruction mode, and comparison mode.

Exhibits visualized in Game Engines. In contrast the the former used X3D viewers much of the recent work on virtual museums and three-dimensional exhibitions uses game engines for displaying virtual artifacts. This is mostly due to the superior visual quality, which can be achieved through the use of such high-end software. Two representative approaches here are, first, an adaption of the Torque3D engine with scripted interaction and an adapted level editor for generating the exhibition layouts by Mateevitsi et al. [MSLV08] and, second, the VEX-CMS from Chittaro et al. [CIR*10], which is based on OGRE [Tor13] and features a custom application, which uses game concepts for exhibition planning.

All in all, game engines could be used for streamlining the production of visualizations in the Cultural Heritage domain, but they have not been developed for such a task and so there exist some shortcomings:

- their authoring tools are proprietary and not tailored for inexperienced users, which are common in the domain of Cultural Heritage,
- they have not been developed for displaying high-resolution scanned models with level of detail mechanics,
- the long- or even mid-term sustainability may become a problem, and
- programming as well as modeling software like Autodesk Maya [Aut13c] is usually required for content creation.

6.4.2 The Designer Mode

The designer mode of the GML Compositor (see Section 4.4) incorporates the full set of modeling, scene graph and animation operations available in this system. In this mode, the designer creates scene templates that are later filled by curators in the curator mode. The tasks to create a scene template are manifold and will be discussed in this section.

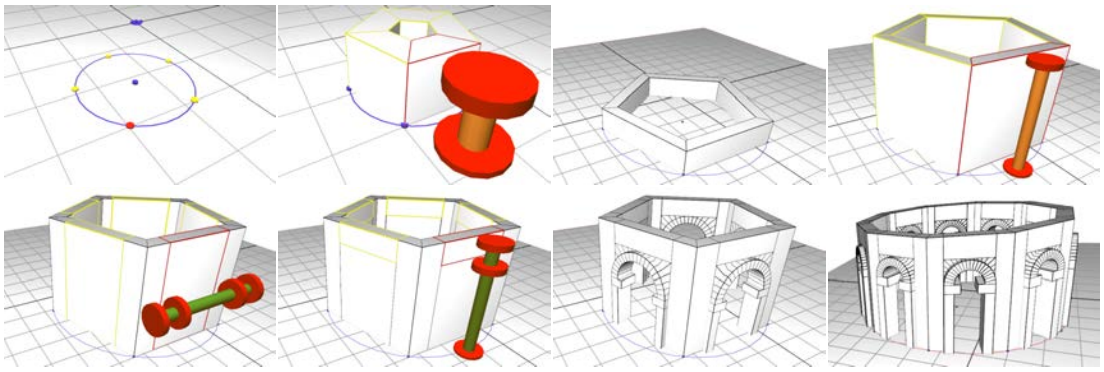


Fig. 6.34 Showcase of the integrated parametric modeling engine of the GML Compositor. Scripted assets like arches and windows are imported from a separate asset library. Step by step a simple parametric archway is generated with only a few clicks. Such model can serve as backdrop for the visualization of virtual artifacts. (image adapted from Zmugg et al. [ZTH*12])

Creating the Scene Backdrop. With the use of the procedural modeling operations of the GML Compositor (see Section 4.4.3) backgrounds can be created easily. The parametric assets used for several reconstruction tasks serve for appropriate backgrounds for historical artifacts. Figure 6.34 shows the application of the integrated modeling engine of the GML compositor to create a simple parametric building featuring several archways.

Creating the Scene Hierarchy. The main task is of course the creation of the scene layout. The scene graph concept is ideal to describe hierarchical scenes. The relative positioning of one object (child) relative to another (parent) results naturally in a hierarchy of transformations. This hierarchy is best visible in animations.

The scene graph itself can be generated using the operations and interactions described in Section 4.4.4. Different libraries within the GML Compositor, such as the compass and ruler library from Wolfang Thaller, can be combined with scene graphs to create regular layouts, e.g. distributing scene graph nodes sharing the same parent on a circle (see Figure 6.35(a)).

Placing Objects and Cameras. Each scene graph node can be associated with an object. These objects can either be three-dimensional models from the repository, cameras or special drop targets, which are for the curator to use. Placing these objects does neither affect the hierarchy nor the functionality of the node. The node widget just resizes itself dependent on the size of the inserted object to further allow easy interaction.

Drop targets are numbered attachments for scene graph nodes. They mark spots for the curator to place assets from the repository. Drop targets can also be created procedurally, so that their exact number is configurable in curator mode. Drop targets have been placed in a scene using compass and ruler functionality in Figure 6.35.

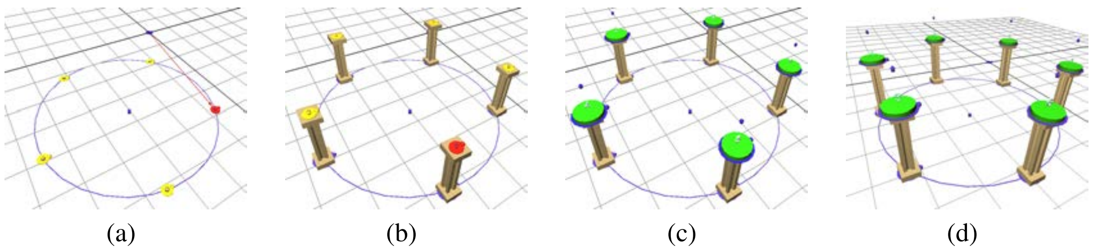


Fig. 6.35 Procedurally generated drop targets. The specific number of drop targets can be configured later in curator mode. The n-gon is constructed through the integrated compass and ruler library. (image adapted from Zmugg et al. [ZTH*12])

Cameras specify pre-defined views that permit directing the attention of the user. Each camera stores a view direction, which can be set by the user interactively. The position and orientation of the camera are defined through the scene graph. The camera parameters can also be defined relative to a (parent) object in a way that this object remains in view even when it moves. A camera setup is shown in Figure 6.36.

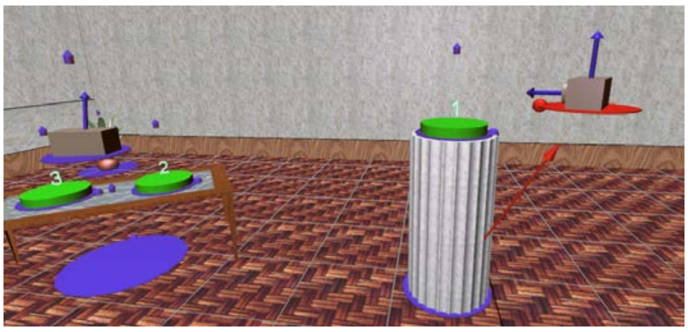
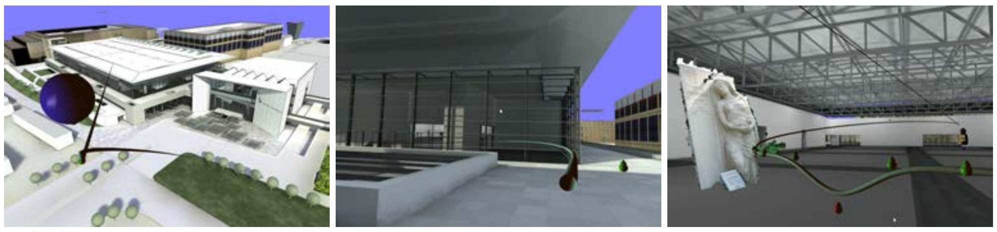


Fig. 6.36 Camera placement for pre-defined views. The camera (right) is a child of the pedestal node (indicated by the red arrow). So the object on the pedestal (drop target number 1) remains in view even when the pedestal is moved. (image source: Zmugg et al. [ZTH*12])

Defining Animations of Objects. Animations can be defined for all scene graph nodes as described in Section 4.4.4. Animating nodes with attached cameras allows for smooth view transitions. For flight-through animations the system provides splines along which a camera can move with the view in flight direction. The control points can be edited in curator mode to adapt the flight path based on the exhibits (see Figure 6.37).

Defining the Functionality of the Curator Mode. The designer has full control over the scene. The designer can change many things in the scene that the curator would not want to touch anymore. Especially large scenes have a confusing number of options and parameters. Therefore, the designer can define which parameters and options are accessible to the curator


(a)

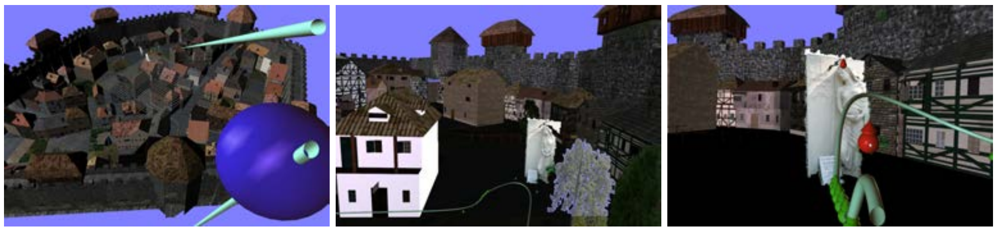

(b)

Fig. 6.37 Fly-through animations along editable splines. The animation was developed using a modern architectural site (Frankfurt fair, (a)) with inserted scanned assets. Since the spline is editable, the scene template could quickly be adapted when the medieval town model (b), which was generated with Esri's CityEngine [Esr13], became available. (image source: Zmugg et al. [ZTH*12])

and which parameters are fixed and inaccessible. Widgets corresponding to inaccessible parameters are, consequently, rendered invisible to reduce the number of objects on the screen. Limiting the configuration options not only shields the curator from accidentally doing harm, but also makes the work of the curator simpler, more targeted and efficient.

The technical basis for this configurability is the central code graph of the GML Compositor that contains all scene parameters. In essence, a Boolean attachment is added to all the values in the graph. This attachment defines whether the corresponding widgets that are used for editing this node are visualized or not.

6.4.3 The Curator Mode

The curator mode shall enable curators, which are not so experienced with handling three-dimensional content, to produce compelling three-dimensional scenes. Scene templates can either be very simple or fairly complex. In the simplest case, they contain only a fixed number of drop targets on which the curator drags artifacts. One can also imagine that such simple scenes can be filled automatically with information provided in e.g. an external document. More flexible scene templates may offer more configuration options, giving more freedom and control to the curator.

There are mainly four tasks for the curator in the curator mode:

- getting hold of three-dimensional assets and deciding which assets are shown,
- describing the desired scene templates to the designer,
- choosing appropriate scene templates for each exhibit, and
- filling templates with assets and configuring the templates.

Acquiring the Exhibits. Museums usually have a set of assets acquired by them using e.g. photo reconstructions. However, high-quality three-dimensional acquisition requires trained personnel from a photographic department or a scanning company. Alternatively, three-dimensional assets can also be rented or even bought from other museums or companies.

The 3D-COFORM project supports the process of finding the right assets based on a semantic metadata network as described in detail in the work of Doerr et al. [DTT*10]. The right assets for the planned exhibition can be found on a semantic basis rather than a purely formal basis, i.e., not just by period or size, but also by style or manufacturing method. It all depends on the metadata available. Once a suitable collection is found, it is grouped to facilitate the further steps of asset production (see Figure 6.38).

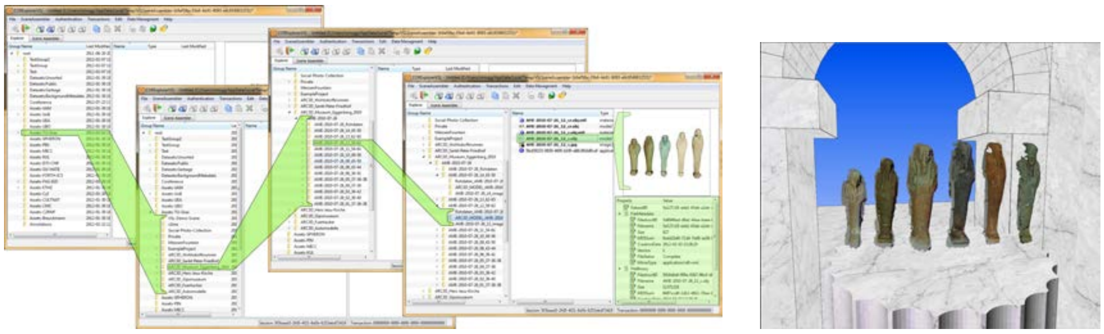


Fig. 6.38 Workflow of a curator selecting assets. The system allows navigating through the group hierarchy of the distributed asset database. The curator chooses the assets for the exhibits and drags them onto the pre-defined drop targets. (image source: Zmugg et al. [ZTH*12])

Use of Scene Templates. After deciding which assets are about to be shown it is necessary to decide how these assets are presented. Depending on the exhibit, different ways for presentation are possible, for example, an explosion view, a small animated story, or a walk-through animation. The curator defines the required functionality for each exhibit and the designer then prepares appropriate scene templates that meet the look and feel in terms of color, layout and design of the real exhibition.

When everything is available, the curator can start filling the scenes to create the digital exhibits. For each showcase, a suitable scene template is chosen and is filled with the digital artifacts (see Figure 6.39). Depending on the skill and requirements, the designer can set certain parameters of the scene to be available for the curator. This way the curator can adapt each template to his needs, like adapting the amount of drop targets (see Figure 6.35), changing camera views, or fine-tuning the position of exhibits and other three-dimensional objects in the scene.

6.4.4 The Presentation Mode

The final step is viewing the result in a separate standalone application that can be executed on displays throughout the museum. The presentations mode is such a standalone viewer application for fullscreen viewing. The viewer is available as Qt widget and can therefore be used

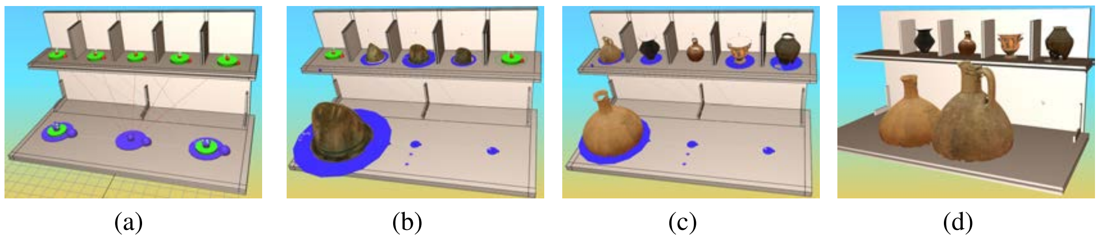


Fig. 6.39 A scene template for object comparison is re-used. The scene template with drop targets (green) was created by the designer (a). It is filled by the curator via drag & drop with helmets (b). It can be re-used with ceramic pots (c). In the presentation mode users can select a pot from the shelf for an animated direct comparison of the two shapes (d). (image source: Zmugg et al. [ZTH*12])

within any Qt-based application and can be embedded in an HTML page in Qt-enabled web browsers as well.

In terms of user interaction, a generic approach is followed. Besides mouse and keyboard input, user events can also be triggered over the network by sending strings via a socket connection. This enables a wide variety of input devices, even gesture recognition can be used for input. Especially gesture recognition using, for example, the Microsoft Kinect Sensor [Mic13b] is a relevant topic because mouse, keyboard, or even touch screen may not be available in museums, for example, due to health concerns.

6.4.5 *Use Cases realized with the GML Compositor*

With the presented system, which is composed of three distinct modes as showcased in Figure 6.40, three distinct use cases have been realized as proof of concept. First, an animated fragment reassembly with alternative assembly hypotheses as an example of a custom-made animation for high-quality content (see Figure 6.41), second, a re-usable scene template for flythrough animations, used with two different models (as shown in Figure 6.37), and finally the comparison use case with two similar objects that are brought to the same pose to highlight the difference (see Figure 6.39).

6.5 V2me - Virtual Coach Reaches Out To Me

The increasing loneliness in Europe's aging population is a severe problem. The goal of the V2me project [V2m13] is to combine real life and virtual social network activities to prevent and overcome exactly this loneliness. Through the activities done within the system the joy of life should be enhanced for all participants. V2me encourages people to continue their participation in in society, to share their experiences and acquired knowledge, and above all, to stay mobile and cognitively agile. An important part of fulfilling this goal takes the *virtual coach*, a three-dimensional avatar that communicates with the user. The target groups are on the one hand young-old individuals (65-74 years) and on the other hand older generations (75+ years). In general, the main goals for these two groups differ slightly. The younger target group should be prevented from feeling lonely and the system should intervene in the loneliness that got hold of older generations.

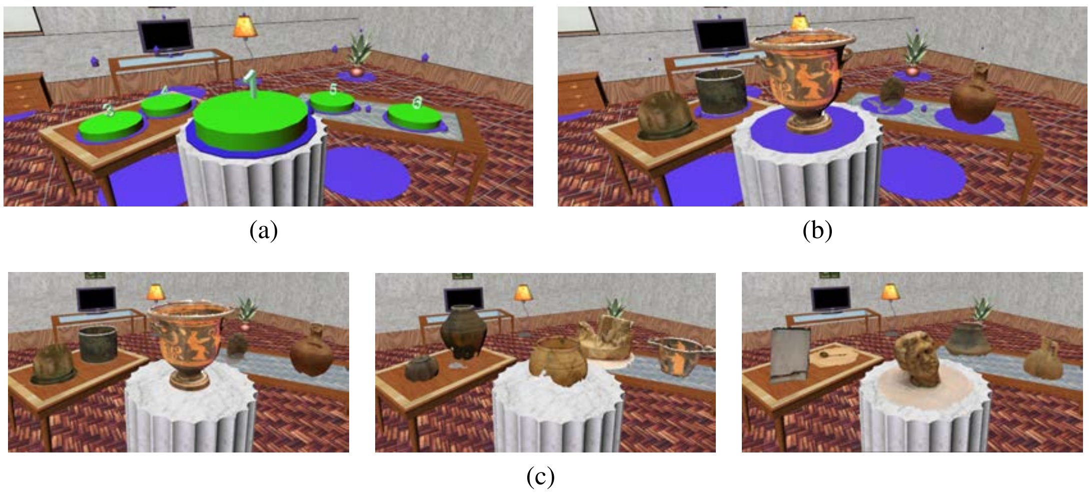


Fig. 6.40 A simple scene template illustrating the looks of the separate modes. In the designer mode (a) drop targets are placed in the scene, which are then filled in the curator more (b) with digital artifacts. The position and orientation of the artifacts can still be changed, which is indicated by the still visible widgets. In the presentation mode (c) no changes are possible anymore because it sole purpose is visualization of the created scene. Three different set of artifacts are positioned in the same template, which emphasizes the re-usability of scene templates.

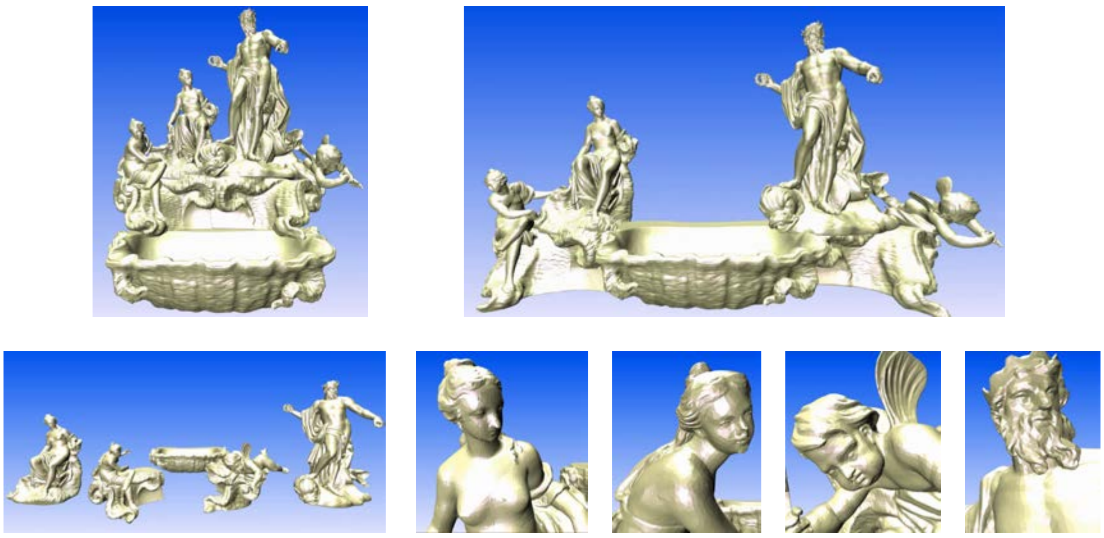


Fig. 6.41 The Meissen Fountain. The five parts can be arranged in two possible ways (top left, top right). An animation can show an interesting transition between both hypotheses, as well as close-up views of the faces. (image source: Zmugg et al. [ZTH*12])

V2me is based on another project called A^2E^2 [A2E13]. This project focuses on virtual coaching of elderly in general. The character model of the virtual coach is taken from this project. This section gives an overview of the system and afterwards focuses on the procedural techniques used to realize the V2me system. The project has also been explained in our work [BCZ*14].

6.5.1 Overview of the V2me Project

This section intends to give an short overview of the V2me system by explaining the single components and how they work together. The V2me system (see Figure 6.42) generally consists of three main components: The *Home Platform*, the *Mobile Platform*, and the *Web Platform*. Furthermore, the system provides two editors for content creation. These editors store their generated data on the provided servers.

My contributions in this project concentrate on the Home Platform, the content editor, as well as the user avatars. All of these system parts are based on GML and the GML Compositor. Thaller provided his assistance in the development of the Home Platform and the content editor.

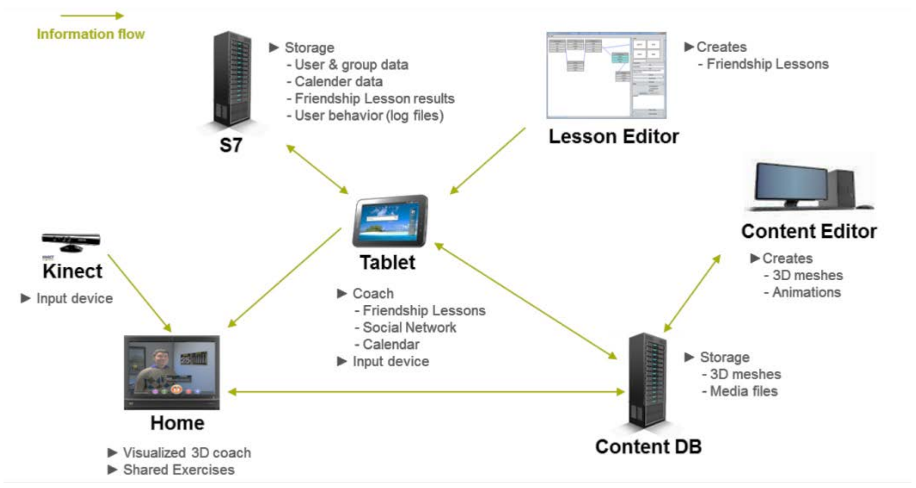


Fig. 6.42 System architecture of the V2me infrastructure.

The Home Platform.
The Home Platform is a stationary system in the premises of the elderly people. It features a single big display used for visualizing a three-dimensional scene in which the virtual coach resides (see Figure 6.43). The main task of the Home Platform is to augment all actions done by the user.

The Mobile Platform.
The Mobile Platform is a touch enabled hand-held system, usually a tablet, which is used as input device. This system is the communication interface between the user and the virtual coach and acts as remote control for the Home Platform. Furthermore, communication with other users is possible with social network services as well as video call services. Intuitive touch gestures help the elderly users to get used of the system very fast.

The Web Platform.
The Web Platform is a web portal that combines multiple web applications into a single view. The purpose of this platform is to connect the relatives of the user with the user and the care givers. Through this system relatives and the care givers can add events to the schedule of the user. The Web Platform, furthermore, hosts all necessary data, such as scripts and three-dimensional models.

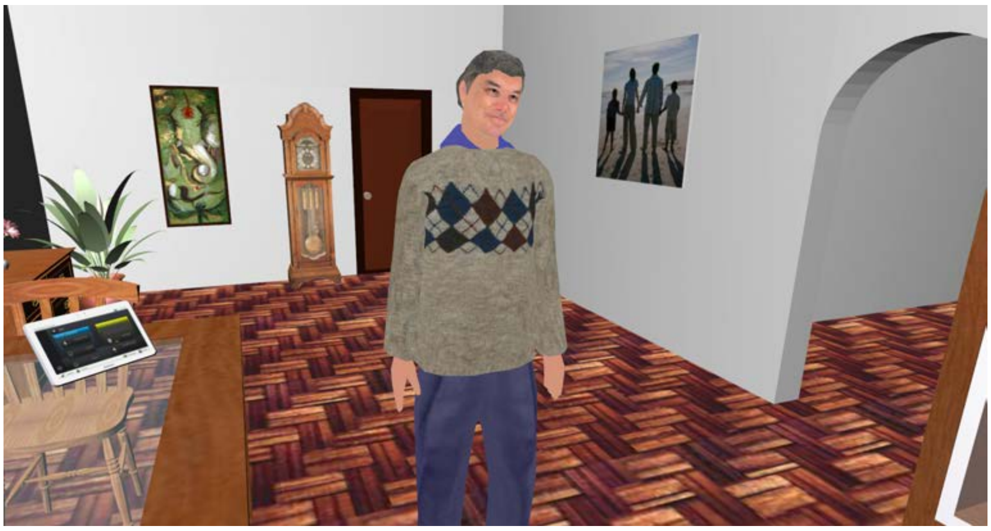


Fig. 6.43 Virtual coach in an environment visualized by the Home Platform. For the creation of these environments a editor, which is based on the GML Compositor, is used. (virtual coach avatar source: the A^2E^2 project [A2E13])

The vision is that elderly people have the Home Platform and Mobile Platform setup installed at home. Both of these platforms are connected to the Web Platform, to which the users have no direct access.

The Virtual Coach. The virtual coach is the core of the developed Ambient Assisted Living solution. The virtual coach is a three-dimensional animated character with text-to-speech ability. The coach is permanently present on the Home Platform and gets in touch with the user on demand through the Mobile Platform. He fulfills three main roles:

Mentor. For the virtual coach to become accepted he has to build up a relationship with the elderly user. The virtual coach acts as the system representation and has more information about the user than vice versa.
Tutor. The virtual coach guides, motivates and supports the user in various aspects, such as giving positive feedback for actions of the user.
Expert. The virtual coach also collects information from various sources, such as weather information. This information is presented to the user in an understandable way.

Facial and body animations as well as the text-to-speech engine of the virtual coach can be triggered independently by the Mobile Platform. So, for example, upon a successful completion of a task by the user, the Mobile Platform triggers a motivating animation of the virtual coach (see Figure 6.44).

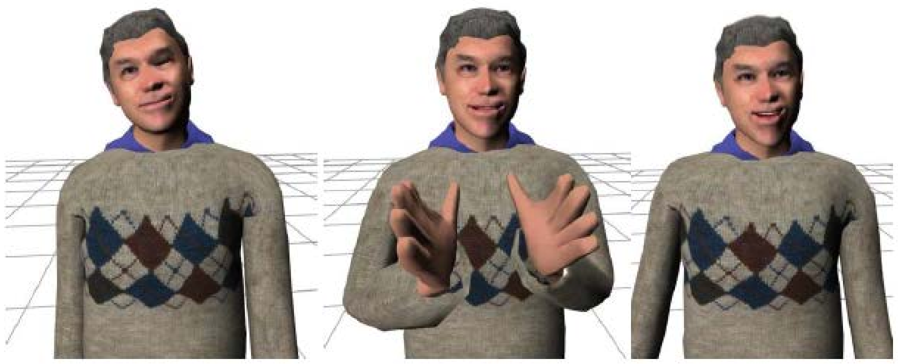


Fig. 6.44 Different animations of the virtual coach that can be triggered by the Mobile Platform based on actions done by the user. (virtual coach avatar source: the $\mathsf{A^2E^2}$ project [A2E13])

The Friendship Enrichment Program. One important use case of the system is the so-called *friendship enrichment program* [Ste01]. V2me offers a virtual and individual version of this program through the so-called *friendship lessons*. These lessons are specially designed by psychologists to teach people how to make new acquaintances and, above all, friends. Especially elderly people need these lessons because they are not provided with natural environments like school or work, where friends are met easily. The friendship lessons include tasks like “talk to a stranger” to develop the skills needed to succeed. The virtual coach is responsible for guidance in these lessons.

Editors integrated into the System. The V2me system provides two editors. One editor, the *lesson editor*, is designed to create friendship lessons and other daily routines for the elderly people. The second editor is the so-called *content editor*, which is based upon the GML Compositor. The usage of the GML Compositor in the domain of Ambient Assisted Living will be the main focus in the upcoming Section 6.5.2.

User Representation within V2me. A very important part is also the way user themselves are represented in the system. One desired goal was also to examine the acceptance of virtual three-dimensional self-representations of the user. Three-dimensional avatars, as used by different role play games or consoles, are not suited for a representation in the system. It is really important that the virtual avatar should look like the human it is impersonating. Therefore the Microsoft Kinect Sensor [Mic13b] has been used to record small three-dimensional videos of certain activities of the user. These videos can be recorded for different *behavior slots*, like “success”, “hiking”, or “dancing”. This way, every user has their own personalized set of animations, which are visualized in appropriate situations. Through this, a virtual self-representation of the user that is interacting with the virtual coach on the Home Platform can be visualized (see Figure 6.45).

Unfortunately, due to time constraints, this feature did not make it into the first project prototype. Future ideas include using these personalized animations to send invitations for activities to other users in the network.

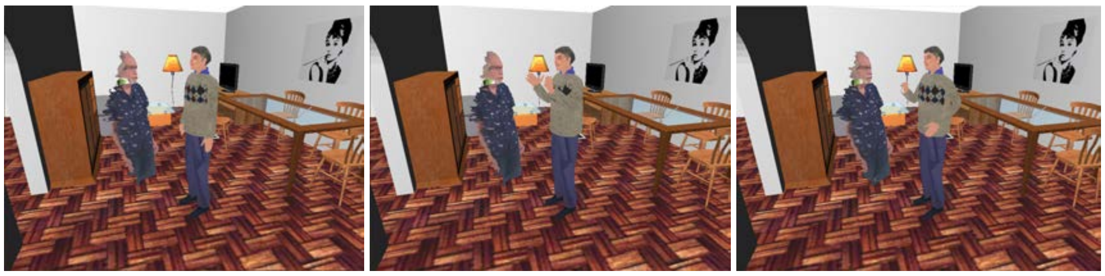


Fig. 6.45 The self-representation of the user is displayed together with the virtual coach in an environment in the Home Platform. The user avatar was generated through use of the Microsoft Kinect Sensor [Mic13b]. (virtual coach avatar source: the A^2E^2 project [A2E13])

6.5.2 Procedural Modeling for Ambient Virtual Coaching Applications

The content editor is again realized through the GML Compositor (see Section 4.4). The editor used within the V2me system is essentially the same that has been proposed for the Cultural Heritage domain (see Section 6.4), excluding the digital repository. The goal, however, is completely different. The content editor should enable non-experts to create three-dimensional content in form of environments that the elderly persons feel comfortable with. This user group includes care givers, as well as the elderly persons themselves. The editor provides all the tools for the procedural modeling engine, the procedural scene graphs and the animations of those. By combining these tool sets, three-dimensional animated stories, which can be personalized for individual persons, can be generated to support the friendship lessons. These animated stories include the virtual coach and his interaction with the elderly person.

Building Non-Static Animated Environments. Initial creation of environments is done using the designer mode discussed in Section 6.4.2. The procedural modeling engine is used to create backdrops, while the procedural scene graphs are used to structure the distribution of objects that are placed in the environment. The focus lies mainly on indoor environments, so a wide variety of appropriate objects (taken from Trimble 3D Warehouse [Goo13d]) is available. The avatar model for virtual coach is also part of the scene graph and needs to be positioned in the scene.

Animations can be defined using the mechanisms described in Section 4.4.4. Additional to the animations of scene graph nodes, body and facial animations of the virtual coach can be set for specific time frames too. So if the position of the virtual coach changes in a time frame, the "walking" animation of the three-dimensional character can be activated to provide a realistic and consistent story. Cameras can also be placed and animated in the scene. Furthermore, they be set active for certain time frames to highlight different places or actions in the scene. Figure 6.46 shows four snapshots of a camera view transition as a part of an animated story that explains certain tablet functionalities to the user.

Adaption of the Curator Mode. The curator mode can be used in the Ambient Assisted Living context in a slightly different way. The curator mode was designed to provide a museum curator with a way to make final changes to the scene layout. The scene is defined through a scene template beforehand, and the curator can fill the scene with virtual artifacts. For this

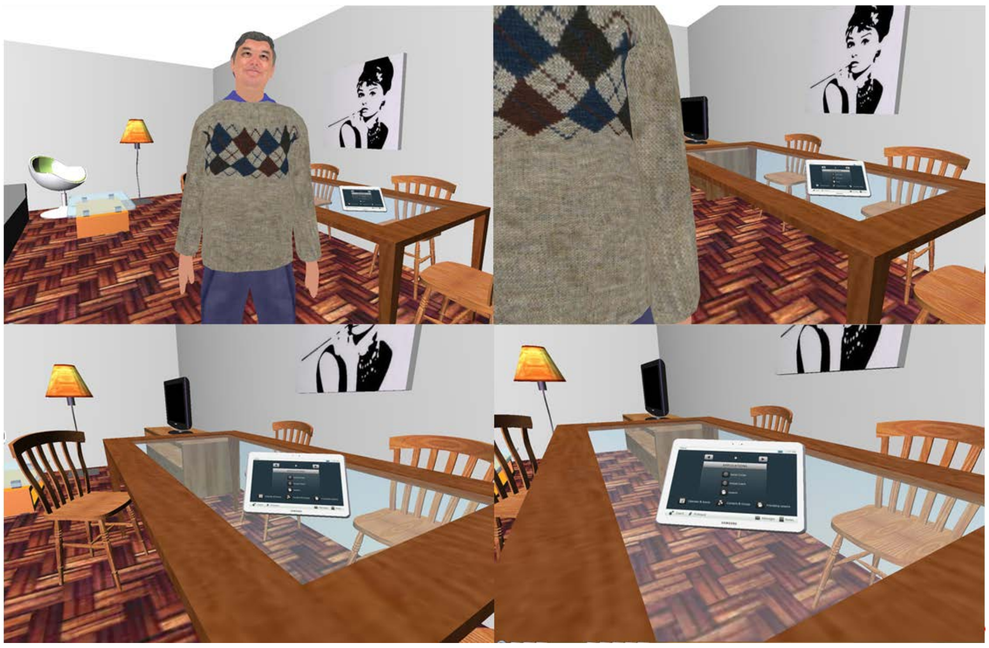


Fig. 6.46 Four snapshots of a camera flight animation that changes from a view of the virtual coach to a close-up view of the tablet on the table. This is part of an animated story that explains certain tablet functionalities to the user. (virtual coach avatar source: the A^2E^2 project [A2E13])

task, the configuration options are deliberately limited so that the curator is shielded from accidentally doing harm to the scene, and to work in a more targeted and efficient way.

Within V2me there is no curator, who changes positions and adapts exhibits in pre-defined scenes, but the situation is similar. Care givers can design environments and the animated stories within them. Afterwards, elderly people may want to change the positions of objects in the scene to personalize the environment. For the animations, which tell a story through the procedural environment, to remain the same, the configuration options of story-related animations and objects are not available. Drop targets could be utilized to provide even more customization options, i.e. additional paintings could be placed on the walls, or more or less chairs could be positioned around a table. Figure 6.47 shows three variations of furniture placement of a living room. Within the V2me system this mode is called the *personalization mode*.

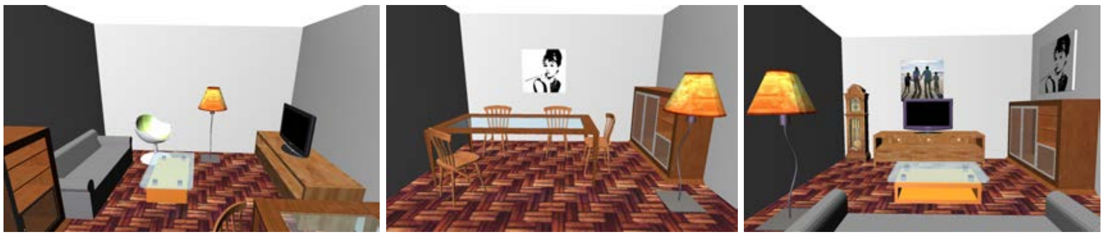


Fig. 6.47 Three variations of different furniture placement in the living room template for the Home Platform. Users can arrange the single furniture objects as they like to meet their personal desires.

The Home Platform and the Interaction with the Mobile Platform. The Home Platform visualizes the environments created in the editor, which makes it equivalent to the presentation mode in the context of virtual museums. The Home Platform, additionally, provides a network interface through which registered animated stories, animations of the virtual coach, as well as the text-to-speech engine of the virtual coach can be triggered. This interface is used by the Mobile Platform to trigger specific animations and events based on the user's actions or at specific times.

6.5.3 Evaluation of the V2me System in Terms of Procedural Modeling.

An evaluation of this system through a user study with focus on procedural modeling has been done in Amsterdam by Roelofsma and Moeskops. This section focuses on the results in terms of procedural modeling of this study.

Setup and Execution of the User Study. Using a living lab setting especially for elderly persons, seven subjects have been selected for the study with this system. The participants are living independently and alone. Subjects have been selected based on their score on the De Jong Gierveld Loneliness Scale [GT06] (scores should be moderate to high) and on their openness to technology (scores should be high). The participants included three women and four men, ranging from 64 to 77 years in age. The subjects received an instruction session after the system was installed in their homes.

The loneliness intervention lasted for two months. Afterwards, they received the loneliness and openness to technology questionnaires again, and a semi-structured interview was held about their experiences with the Home Platform. In this interview they had been asked questions on what they thought of the Home Platform and the interior of the dwelling, and what they would like to see changed. The functionalities of the personalization mode have not yet been included in this user study. The goal was to assess the interest in using the procedural features of such a mode.

Response of the Elderly Participants. This part focuses on the interpretation of the answers that have been given to specific questions of the questionnaire and what implications these answers have to the V2me system. Because the participants are usually not used to terms like procedural modeling, questions had to be designed properly to assess their interest in such features.

Asked about their general opinion of the Home Platform, the participants liked the system, although there was a variety in why they liked it. Each participant showed to have their own preferences when they had been asked how they liked the apartment of the coach. One subject mentioned that *"It is very beautiful, very well made. [...] I like the interior a lot."*, whereas other comments included *"I love how the parquet has been made"* and *"I like looking at him. The way he shakes his head."*.

Important are the answers on the question what elements are missing from the virtual environment. These answers are the main indicator for an interest in procedural features. The responses here varied from *"The house lacks a lot of stuff."* to *"It is too crowded."*; both of these extremes indicate that the users want to change the environment, may it be through adding or removing objects. For example, some participants would have preferred the coach to reside in a more comfortable environment. They would like to add a couch, and a little lamp, so that he can sit down and read. Whereas others concluded that *“He needs a stereo set, a TV, and a leather couch.”*, or *“He could use a fridge, or a stove in his kitchen.”*. Based on these answers it is justifiable to assume that an interest in changing the environment – and therefore in procedural modeling features – is present and the personalization mode should be considered for future studies.

Synopsis

In this chapter I presented all results that have been acquired through the use of the presented procedural modeling techniques. The majority of these results have been achieved by me. The remaining results were either achieved in collaboration with others or by students I helped supervising.

The architectural results mainly showcase advantages of the presented split grammar extensions, by either describing a high quality example such as the Rialto Bridge and the Great Wall of China, or by describing a more general class of buildings that are suited for mass generation.

The GML Compositor has been utilized in two ways. The split grammar inspired modeling tool kit has been used to create reconstructions of iconic façades of the Louvre. This reconstruction was performed to demonstrate and assess the capabilities of the software and indicate directions of future work. The second application of the GML Compositor, in which I was heavily involved, has been the generation of animated procedural environments. These have been utilized to provide animated museum exhibits that can adapt to different situations and to provide animated environments for story telling purposes to motivate lonely elderly people to become socially active again.

Chapter 7
Discussion and Conclusion

Contents

Abstract. To conclude this thesis, I will take the chance to discuss potentials and issues of procedural modeling in this last chapter. This discussion will then proceed and focus on the three representative problem areas of this thesis, namely the understanding of shape spaces, the reconstruction of architecture, and interactive procedural modeling. The contributions and limitations of the presented techniques will be reviewed and possible future work in these domains is explored. Finally, this chapter gives an outlook on how the vision of Seidel will be pursued in the future and what importance this thesis has in this regard.

7.1 Potentials and Issues of Procedural Modeling

Understanding shapes is the key to procedural modeling and consequently the key to fulfill the vision of visual computing. When a shape is understood, all modeling operations can be extracted to create infinite valid variations of that shape. Based on the work done in this thesis, I will take the chance to discuss potentials and issues of procedural modeling in this section.

Refactoring and Reinterpretation. In a procedural model parameters can be defined explicitly, meaning there is direct access to the value, or implicitly, which means that the value of this parameter is dependent on values of others. Some of these implicit dependencies are not even known beforehand and will only surface during development. An illustrative example for this is a procedural model of a staircase. For a given room height, giving access to the parameter that controls the step height implicitly defines the value for the amount of steps and vice versa. Which parameters are made accessible depends on the application. To find the ideal description for a procedural model, refactoring is often necessary. However, there is not always an optimal solution, but a set of alternatives, each with their own advantages and disadvantages. The main issue here is that it is very hard, and often impossible, to combine two different descriptions from the same shape family to express all shapes that have been expressed by the single ones.

Imagine a procedural model of a house with indoor environments. A question that may arise is how to model the walls that separate the individual rooms. Let us imagine a wall that connects room A with room B. When the wall is realized as a single object belonging to one room it is adjacent to (without loss of generality let us assume it is room A as in Figure 7.1(a)), placement of doors, for example, is an easy task when performed outgoing from room A. However, the placement of objects on one side does influence the other side. Furthermore, objects placed on the shared wall in room B have to be generated corresponding to a wall, that does not belong to the room itself, which is contradictory. Another way to realize this is to introduce “half walls”. The wall between room A and B is split into two parts and to each room one half of the wall belongs (see Figure 7.1(b)). This way, all objects can be placed on a wall that corresponds to the right room, but connecting structures like doors need to assure that they are placed exactly in the same spot on both halves of the wall. Both methods are advantageous in one regard, but not ideal in another. A combination of both (illustrated in Figure 7.1(c)) is conceivable and realizable through an appropriate design in this case, but this may not be true for all cases.

To conclude, just taking the first alternative that comes to mind will often turn out wrong. The problem, as well as all alternatives, need to be analyzed to take the one that meets the needs best. However, one should always be prepared for refactoring.

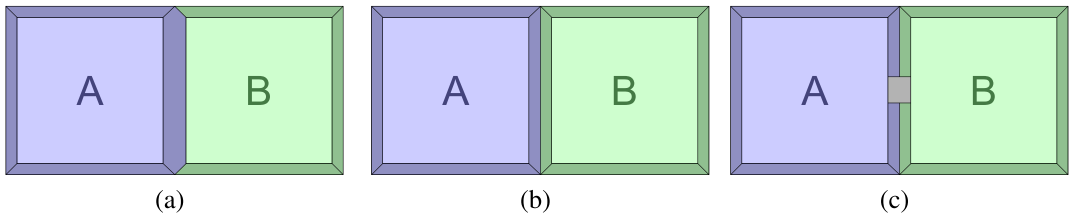


Fig. 7.1 Ways of representing a wall between two rooms A and B. The wall can entirely belong to one room (a) or can be shared between the two (b). Both methods have different behavior in terms of generating connecting structures (doors and windows) and associating elements (furniture) to the wall for the separated rooms. A combination of both (c) can be achieved through an appropriate design.

Importance of Shape Understanding for Inverse Procedural Modeling. Nowadays, inverse procedural modeling is a very prominent topic, but it is also a problem that is very hard to solve. What is actually often forgotten and rarely mentioned is that the design of procedural models in a forward manner is a prerequisite to make inverse procedural modeling techniques even work. This important step is necessary to build up knowledge about the modeling domain and extract the operations necessary to model objects from this domain. Without knowing these building operations inverse procedural modeling techniques are doomed to fail. Just imagine the procedural window building blocks that have been designed with the help of the Generative Fact Labeling method (see Section 5.2). This library of procedural window building blocks is an ideal stepping stone for inverse procedural modeling of windows from photographs, but without such a modeling library the task of fitting three-dimensional models to exemplars depicted in photographs is far more complex.

Comparison to Structured Shape Editing Techniques. Procedural modeling is a very versatile technique. It is more expressive than data-driven structured shape editing techniques that only rely on combining parts of a limited data set. A single procedural models can encode a whole object family on its own. By providing a set of parameters, far more different models can be achieved than just be interpolating between fixed model instances. However, there is still the overhead of generating the procedural models, which is no easy task. But the same is true for structured shape editing techniques, where an appropriate data set needs to be found and learned.

Another shortcoming of structured shape editing techniques is related to the amount of available exemplars. While procedural modeling techniques show that describing object families from a very small set of exemplars is plausible and possible, learning based on very small sets of exemplars is rarely successful. Procedural modeling techniques can even be applied to single instances (as showed in the example of the Rialto bridge in Section 6.2.2), but automatic learning from single instances does hardly seem worthwhile.

Procedural Modeling Editors in Industry. Another important fact that emerged during the research done in this thesis is that there is a high demand for procedural models in industry. Especially the procedural wedding ring task (see Section 6.1) showed that there is a need for procedural modeling editors for mass customization. Editors, such as the GML Compositor, may play a huge role in industry in the future. This underlines the importance of forward modeling through procedural descriptions. A lot of manpower is necessary to meet this demand for procedural models in industry. Aside from wedding rings, there are far more different domains where people like to have their individual personalized items, among which are furniture items, car keys, or accessories. For now this may only concern luxurious items, but in the not so far future this customization should be available for everyone. To realize this procedural modeling in combination with rapid prototyping through three-dimensional printers is the best answer.

The Limits of Procedural Modeling. Everything that follows order and rules can be realized by procedural modeling. This usually applies to all man-made objects that can be described by a manufacturing process. Completely random patterns that do not show any signs of rules are the limit of procedural techniques. Human products are never just random; they are generated through ideas and thoughts. This, however, implies that we are able to understand the thought processes of every human, which is certainly not the case. Especially, for many kinds of art, it is hard to convey the implicit knowledge and intuition used to create the artwork to other people. Even though the foundations of shape grammars lie in the interpretation of art [SG72], paintings, sculptures, and these kinds of art feature their own kind of randomness that is hard to describe. These features are often depicted as *artistic freedom*.

Randomness does actually not exist, it is just a statement that there are mechanisms that we do not understand yet. As Albert Einstein said: *"God doesn't play dice"*. This means for objects that are not made by man, all of them, even though some may seem completely random, underlie the rules of physics and nature and can therefore be represented by mathematical models once these are understood. However, even though modeling all understood mechanisms down to a cellular level could describe shapes, it is just too complicated and low-level. Approximations through high-level operations are made to achieve satisfactory results with less effort.

7.2 Conclusion and Future Work

I introduced a desirable concept in the introduction. This concept includes that the three-dimensional physical world will be step by step realized as digital editable three-dimensional content, which allows three-dimensional content creation for everyone. This vision has been formulated as the goal of visual computing by Seidel. To implement this vision, the state of the art of computer graphics, in particular procedural modeling, has to be advanced further.

In this thesis I have presented new ways and improved established methods to express complex shape domains in a more efficient way. The applied research I have done, in collaboration with my colleagues Krispel [Kriar] and Thaller [Thaar], in this regard helped in the development of these methods and, furthermore, extended the potential of them. The task to map the real world to a editable digital counterpart is such a big undertaking that is not realizable in the context of a single Ph.D. thesis, therefore, I focused on three representative problem areas in the domain of procedural modeling (see Section 1.4). To conclude this thesis, I will discuss the progress, the improvements, as well as potential for future work in these three categories.

7.2.1 Understanding Shape Spaces – Creation of Procedural Models

The generation of procedural models is a very important task. Procedural models – once created – save a lot of time for generating selected instances of a shape family. The generation of such a description is the hard task. This task has been illustrated in Section 5.2 on the domain of architectural windows together with a formulation of the Generative Fact Labeling method. This method has been developed by Thaller, Havemann and me and encapsulates the approach of designing a procedural model out of a set of exemplars in a formal way. A key concept here is factoring out features. While a use case may seem too complicate for an easy procedural description, it can become easily manageable through factoring out one feature after another. This was demonstrated by factoring out profiles and moldings from the domain of windows. This reduced the complexity of several windows elements, which afterwards could be described in an efficient manner.

My opinion is that through this extremely generic method a paradigm shift in how procedural models are generated could happen. This method helps to create procedural libraries with high-level modeling operations to describe any domain of man-made shapes. However, this generality is also the weakness of this method. Although the method is effective, but for this efficiency, however, much sophistication is necessary. Nevertheless, this method is the stepping stone for new fields of research and business areas. Together with the upcoming trend of mass customization, procedural models become more and more important to realize personalized items for everyone (like the wedding rings in Section 6.1). To meet this demand for procedural models, efficient techniques and trained personnel are necessary to obtain procedural models faster. In the not so far future, firms like IKEA will not advertise their products through printed catalogs anymore; procedural models will take their place to show the different variations of products more clearly.

7.2.2 Reconstruction of Architecture – Architectural Split Grammars

In this thesis two extensions to the state of the art of split grammars are presented:

- First, split grammars on convex polyhedra (see Section 4.2), which utilizes a new non-terminal class convex polyhedra. This extension has been a collaborative work of Thaller, Krispel and me and is an integral part of our theses.
- Second, deformation-aware split grammars (see Section 4.3), which include free-form deformations in the split grammar formalism. This work is mainly attributed to me, but has also been developed in cooperation with Thaller and Krispel.

Grammars, in general, derive much of their power from the ability to re-use rules in different contexts. In case of split grammars, this context is described by the scope of the non-terminal shapes. By generalizing to convex polyhedra, rules can adapt to a much wider range of shapes. Important is that this generalization means never a loss in flexibility, everything that can be done with boxes, can be done with convex polyhedra too. This means box grammar models can be translated in a straightforward manner to models based on convex polyhedra. Although through the volumetric nature of convex polyhedra, volumetric models can be built procedurally without the need to resort to imported pre-modeled assets.

However, in comparison to simple box grammars with no separate geometry, all this added flexibility comes at a cost in implementation complexity. Nevertheless, this implementation overhead is justifiable when compared to the amount of special purpose operations that are necessary to achieve the same expressiveness in box grammars. Furthermore, dealing with convex polyhedra is no more complicated than splitting non-terminal shapes which have geometry that is independent of the corresponding scope in a box grammar (as in the work of Müller et al. [MWH*06]).

The presented extension to standard split grammars to facilitate use of free-form deformations in the procedural modeling process allows deformed shapes to be processed in two ways. The first is a special split operation that calculates distances on the deformed geometry to adapt to changes in space introduced by the deformation. The second is splitting with a straight plane after the deformation. Both kinds of split operations have their rightful applications within split grammars with integrated deformations.

Adjacent walls and roofs of deformed buildings, whose deformations are specified independently, have been joined together by suitable Boolean operations. So far, rules cannot adapt to the results of these operations, which means decorating roof ridges is not possible yet, for instance. The general Boolean operations involved have a significant impact on performance, but are necessary to achieve the results shown in this regard.

Both of these extensions opened up way a for a better description of several kinds of buildings, while keeping the descriptions as simple as possible; they do in no way clutter up the descriptive language, they provide an easy way to access the additional features.

Future Work in Terms of Procedural Reconstruction of Architecture. Interesting future work lies in finding the ideal geometric representation. Such a representation should allow a greater variety of deformations and should eliminate any geometric approximations. Convex polyhedra are here only the first step on a way to the ideal shape description.

The focus of the presented grammar extensions lied in showing single instances of buildings. The fact that all operations are integrated in a standard context-free split grammar means that existing methods (e.g., from Müller et al. [MWH*06]) can be leveraged to create randomized instances of deformed buildings automatically on a large scale, which is important future work.

Interesting further reconstruction tasks, which have been discussed in our group in this domain, include among others a reconstruction of historical Cairo, cargo ships, as well as medieval castles. Further work on procedural descriptions of interiors is also a very prominent topic because the description of complex room arrangements with arbitrarily shaped rooms through split grammars is still an unsolved problem.

However, the high-end goal in the domain of split grammars is the description of modern free-form architecture. This is an example for an especially difficult shape space to describe with high-level modeling operations. While classical architecture features set structures and rules, modern architecture does not have to. It is sometimes pointless to formulate high-level split grammar rules for architecture in general because architects like to violate any rules that can be interpreted in buildings to create something unique. Therefore, the most realistic way to describe architecture in general is to group buildings in different classes and provide formal descriptions and high-level operations for each single one of those. In this regard, great potential and a challenge lie within a procedural description of the iconic buildings of the famous Spanish architect Antoni Gaudi.

7.2.3 Interactive Procedural Modeling – Direct Manipulation Editors

The development of the GML Compositor (see Section 4.4) by Thaller and the extensions made by Krispel and me led to a tool which features direct manipulation of the procedural description on the concrete three-dimensional model. These features are very important to provide procedural modeling without the need for programming. This is essential for visual artists, who are not necessarily experts in programming. However, using data flow graphs to visualize the scene's hierarchy is not ideal either because they do not provide a clear representation for medium- to large-sized models. Therefore, the GML Compositor features highlighting of related parts in the three-dimensional scene. This highlighting clarifies the hierarchy without directly displaying the underlying graph or code that describes the scene. This revolutionary approach will define how non-experts will create procedural models in the future.

The GML Compositor has been utilized in many domains and applications, such as in a reconstruction of parts of the Louvre (see Section 6.2.1), in Cultural Heritage (see Section 6.4), and in an Ambient Assisted Living context within the V2me project (see Section 6.5).

The Louvre use case shows that the GML Compositor enables non-expert users to create more sustainable three-dimensional reconstructions of historic buildings through the procedural modeling approach. Aside from the Louvre reconstruction task, the system is anticipated to be useful for speculating about destroyed and damaged buildings too. New archaeological findings might require adapting the structure or adjusting parameters of a three-dimensional hypothesis, which can be carried out much faster and more efficient on a procedural model than on other model descriptions.

For the latter two applications, which were the main focus of my work with the GML Compositor, hierarchical scene generation through procedural scene graphs was the main topic. To prepare the tool and to provide different functionalities for different user groups, a trisection into modes happened. A mode for experts, who provide scene layouts and limit the configuration options for the second group, which usually consists of experts in other domains. This second group has more experience in how the scene should look in the end, but has, in general, no experience in using general modeling tools. The animated scene is adjusted and finalized by this user group. The last mode then visualizes the animated scene and provides an interactive user interface and is accessible for everyone.

In case of the V2me project, the user study showed that there is a big interest in personalizing the procedural environments. The answers of the participants indicate that a single environment is not sufficient because the priorities and preferences of the participants differ vastly. This shows that procedural modeling needs to be facilitated so that the wishes elderly people have for virtual coaching systems can be fulfilled with ease.

Future Work in Terms of Interactive Procedural Modeling. The GML Compositor is still in its prototype phase and large user studies are still pending. The results achieved so far are promising, but a lot of work is still needed. There are still user interface experiments going on to evaluate how the modeling hierarchy of the model and the parameters can be displayed ideally while keeping the underlying data flow graph hidden. The focus so far was on functionality rather than usability. Therefore, especially with elderly people in mind, all modification options should be accessible in an intuitive way.

For the Louvre use case it is to say that about two thirds of the reconstruction time is spent modeling and specifying fine façade details such as extrusions and profiles. This suggests that there is much potential for improvement. The focus in the development so far has been on a compact set of basic operations and on means to combine them. For example, all the details shown in the reconstruction (e.g. profile extrusions) have been modeled mainly using several single extrude operations that together form the molding running along the façade. While the basic operations are expressive enough to achieve the results shown, the amount of time spent modeling details suggests that developing special purpose tools for different situations, such as profile extrusions, is practical. To create complete reconstructions of historic buildings in the GML Compositor, further operations are still missing or in development. Among these are operations for adding roofs or incorporating scanned material (e.g. statues).

The presentation mode utilized in the museum and Ambient Assisted Living context still misses more realistic rendering, which is planned to be included in the future. The visual quality of the presented results is definitely not yet up to par with the requirements of a museum presentation. The focus of the results presented lies within the concepts; high-quality rendering can be achieved with high-quality scanned artifacts and by using appropriate materials for the geometry.

7.3 Outlook on Pursuing the Goal of Visual Computing

The potential that lies within procedural modeling is huge, but it is not yet utilized in the way to fully exhaust this potential. On the one side, procedural modeling techniques have been increasingly used for content creation for games and motion pictures. If this trend continues, content creation might be entirely procedural in the future. Cities, oceans, forests, all environments can be created randomly in a fraction of the time needed for manually modeling, and every object can be unique and no copied instances will exist. Even animations of characters can adapt to the environment, without motion capturing every single movement. On the other side, procedural modeling provides tools that allow everyone to process and generate three-dimensional content. By following this trend, three-dimensional data will become as easy to process and create as two-dimensional images are today. The development and research of procedural modeling techniques has to proceed to realize these ambitious goals, which are both connected to the vision of visual computing. This thesis is a further stepping stone on the way to realize this vision. The state of the art of several established procedural models has been improved and new techniques have been introduced to ease the task of bringing the physical world into the computer and make procedural modeling universally applicable.

Actually, doing all the analysis during this thesis affected me in how I perceive man-made objects. For example, when I am walking through a street and look at the façades I cannot help but start to begin to analyze them and – in my mind – break them down into the parts they are made of. This implicit knowledge is hard to convey to other people. By transferring it into explicit knowledge through creating procedural models, the matter can be made more accessible to others.

Every man-made shape is described by parameters, and through a thorough analysis these can be found and finally utilized in a procedural model. This, actually, helps me believe that the vision of Seidel is achievable, however, not in the scope of a single Ph.D. thesis. This thesis, therefore, prepares the way and is the first in a series of theses that pursue the realization of this vision.

References

A2E13. A^{2}E^2, Adaptive Ambient Empowerment of the Elderly, 2013. [Accessed: 2013-11-27]. URL: http://www.a2e2.eu/. See pages 195, 196, 197, 198, and 199.

AAAG95. AICHHOLZER O., ALBERTS D., AURENHAMMER F., GÄRTNER B.: A novel type of skeleton for polygons. *Journal of Universal Computer Science 1*, 12 (1995), 752–761. [IIG-Report-Series 424, TU Graz, Austria, 1995]. URL: http://www.jucs.org/jucs_1_12/a_novel_type_of, doi:10.3217/jucs-001-12-0752. See pages 35, 85, and 116.

AC98. AGARWAL M., CAGAN J.: A blend of different tastes: the language of coffeemakers. *Environment and Planning B: Planning and Design 25*, 2 (1998), 205–226. URL: http://EconPapers.repec.org/RePEc:pio:envirb:v:25:y:1998:i:2:p:205-226, doi:10.1068/b250205. See page 24.

AD04. ALEGRE O., DELLAERT F.: A probabilistic approach to the semantic interpretation of building facades. In *In Int. Workshop on Vision Techniques Applied* (2004), pp. 1–12. See page 33.

Ado99. ADOBE SYSTEMS INC.: *PostScript Language Reference Manual*, third ed. Addison-Wesley, 1999. See page 80.

ADS06. AUGSDÖRFER U., DODGSON N., SABIN M.: Tuning subdivision by minimising gaussian curvature variation near extraordinary vertices. *Computer Graphics Forum 25*, 3 (2006), 263–272. doi:10.1111/j.1467-8659.2006.00945.x. See page 59.

AJS12. ANDREWS J., JIN H., SÉQUIN C.: Interactive Inverse 3D Modeling. *Computer Aided Design and Applications 9*, 6 (2012), 881–900. Presented at CAD'12. URL: http://graphics.berkeley.edu/papers/Andrews-II3-2012-06/, doi:0.3722/cadaps.2012.881-900. See page 31.

AK84. AONO M., KUNII T.: Botanical tree image generation. *Computer Graphics and Applications, IEEE 4*, 5 (may 1984), 10–34. doi:10.1109/MCG.1984.276141. See page 36.

AKC88. ARVO J., KIRK D., COMPUTER A.: Modeling plants with environment-sensitive automata. In *In Proceedings of Ausgraph88* (1988), pp. 27–33. See page 37.

Ale65. ALEXANDER C.: A city is not a tree. In *Architectural Forum 122 (1) (part 1), (2) (part 2)* (1965), pp. 58–61, 58–62. See page 21.

Aut13a. AUTODESK: 3D Studio Max, 2013. [Accessed: 2013-11-27]. URL: `http://www.autodesk.com/3dsmax`. See page 15.

Aut13b. AUTODESK: AutoCAD, 2013. [Accessed: 2013-11-27]. URL: `http://www.autodesk.com/autocad/`. See pages 15 and 16.

Aut13c. AUTODESK: Maya, 2013. [Accessed: 2013-11-27]. URL: `http://www.autodesk.com/maya`. See pages 15, 16, 142, 187, and 188.

Bar84. BARR A. H.: Global and local deformations of solid primitives. *SIGGRAPH Comput. Graph. 18*, 3 (Jan. 1984), 21–30. `doi:10.1145/964965.808573`. See page 75.

Bau75. BAUMGART B. G.: A polyhedron representation for computer vision. In *Proceedings of the May 19-22, 1975, national computer conference and exposition* (New York, NY, USA, 1975), AFIPS '75, ACM, pp. 589–596. `doi:10.1145/1499949.1500071`. See page 53.

BCZ*14. BRAUN A., CIESLIK S., ZMUGG R., WICHERT R., KLEIN P., HAVEMANN S.: V2me - Virtual Coaching for Seniors. In *Wohnen - Pflege - Teilhabe - Besser leben durch Technik. 7. Deutscher AAL-Kongress* (2014), VDE-Verlag GmbH, Berlin, Offenbach, p. 5. URL: `http://www.vde-verlag.de/proceedings-en/453574020.html`. See pages xi and 195.

Ben13. BEN FRY AND CASEY REAS: Processing, 2013. [Accessed: 2013-11-27]. URL: `http://www.processing.org/`. See page 16.

BF09. BERNSTEIN G., FUSSELL D.: Fast, exact, linear booleans. In *Proceedings of the Symposium on Geometry Processing* (Aire-la-Ville, Switzerland, Switzerland, 2009), SGP '09, Eurographics Association, pp. 1269–1278. `doi:10.1111/j.1467-8659.2009.01504.x`. See page 71.

BKP*10. BOTSCH M., KOBBELT L., PAULY M., ALLIEZ P., LÉVY B.: *Polygon Mesh Processing*. A K Peters, Ltd, 2010. See pages 47 and 50.

BL09. BORNHOFEN S., LATTAUD C.: Competition and evolution in virtual plant communities: a new modeling approach. *Natural Computing: an international journal 8*, 2 (June 2009), 349–385. `doi:10.1007/s11047-008-9089-5`. See page 21.

Blo85. BLOOMENTHAL J.: Modeling the mighty maple. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1985), SIGGRAPH '85, ACM, pp. 305–311. `doi:10.1145/325334.325249`. See page 36.

BM02. BENEŠ B., MILLÁN E. U.: Virtual climbing plants competing for space. In *Proceedings of the Computer Animation* (Washington, DC, USA, 2002), CA '02, IEEE Computer Society, pp. 33–42. URL: `http://dl.acm.org/citation.cfm?id=791218.791582`. See page 37.

BN88. BIERI H., NEF W.: Elementary set operations with d-dimensional polyhedra. In *Proceedings on International Workshop on Computational Geometry on Computational Geometry and its Applications* (New York, NY, USA, 1988), Springer-Verlag New York, Inc., pp. 97–112. URL: `http://dl.acm.org/citation.cfm?id=53477.53486`. See page 71.

BSly. BBC, SILICON GRAPHICS (SGI): Mummy: Inside story, 2004 July. Exhibition at the British Museum London, http://www.thebritishmuseum.ac.uk. See page 188.

BSK*12. BERNDT R., SCHINKO C., KRISPEL U., SETTGAST V., HAVEMANN S., EGGELING E., FELLNER D. W.: Ring's anatomy – parametric design of wedding rings. In *CON-*

TENT 2012 (2012), Xpert Publishing Services, Wilmington, USA, pp. 72–78. See pages 158, 159, 160, 161, and 162.

Bue93. BUELINCKX H.: Wren's language of city church designs: a formal generative classification. *Environment and Planning B: Planning and Design 20*, 6 (1993), 645–676. URL: `http://EconPapers.repec.org/RePEc:pio:envirb:v:20:y:1993:i:6:p:645-676`, `doi:10.1068/b200645`. See page 24.

BWKS11. BOKELOH M., WAND M., KOLTUN V., SEIDEL H.-P.: Pattern-aware shape deformation using sliding dockers. In *Proceedings of the 2011 SIGGRAPH Asia Conference* (New York, NY, USA, 2011), SA '11, ACM, pp. 123:1–123:10. `doi:10.1145/2024156.2024157`. See page 32.

BWS10. BOKELOH M., WAND M., SEIDEL H.-P.: A connection between partial symmetry and inverse procedural modeling. In *ACM SIGGRAPH 2010 papers* (New York, NY, USA, 2010), SIGGRAPH '10, ACM, pp. 104:1–104:10. `doi:10.1145/1833349.1778841`. See pages 32, 39, and 40.

BWSK12. BOKELOH M., WAND M., SEIDEL H.-P., KOLTUN V.: An algebraic model for parameterized shape editing. *ACM Trans. Graph. 31*, 4 (July 2012), 78:1–78:10. `doi:10.1145/2185520.2185574`. See page 32.

CA11. CALOGERO E., ARNOLD D.: Generating alternative proposals for the louvre using procedural modeling. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences XXXVIII-5/W16* (2011), 185–189. URL: `http://www.int-arch-photogramm-remote-sens-spatial-inf-sci.net/XXXVIII-5-W16/185/2011/`, `doi:10.5194/isprsarchives-XXXVIII-5-W16-185-2011`. See pages 138, 139, and 140.

CB12. CHRISTIANSEN A. N., BAERENTZEN J. A.: Generic graph grammar: a simple grammar for generic procedural modelling. In *Proceedings of the 28th Spring Conference on Computer Graphics* (New York, NY, USA, 2012), SCCG '12, ACM, pp. 85–92. `doi:10.1145/2448531.2448542`. See page 30.

CC78. CATMULL E., CLARK J.: Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer-Aided Design 10*, 6 (1978), 350–355. URL: `http://www.sciencedirect.com/science/article/pii/0010448578901100`, `doi:10.1016/0010-4485(78)90110-0`. See page 59.

CCGP10. CHEVRIER C., CHARBONNEAU N., GRUSSENMEYER P., PERRIN J.-P.: Parametric documenting of built heritage: 3d virtual reconstruction of architectural details. *International Journal of Architectural Computing 8*, 2 (2010), 135–150. `doi:10.1260/1478-0771.8.2.135`. See page 142.

CEW*08. CHEN G., ESCH G., WONKA P., MÜLLER P., ZHANG E.: Interactive procedural street modeling. In *ACM SIGGRAPH 2008 papers* (New York, NY, USA, 2008), SIGGRAPH '08, ACM, pp. 103:1–103:10. `doi:10.1145/1399504.1360702`. See page 35.

Cga13. CGAL, Computational Geometry Algorithms Library, 2013. [Accessed: 2013-11-27]. URL: `http://www.cgal.org`. See page 71.

Chi05. CHITHAM R.: *The Classical Orders of Architecture*, second ed. Architectural Press, 2005. See page 144.

Cho56. CHOMSKY N.: Three models for the description of language. *Information Theory, IRE Transactions on 2*, 3 (Sept. 1956), 113–124. doi:10.1109/tit.1956.1056813. See page 18.

CIR*10. CHITTARO L., IERONUTTI L., RANON R., VISINTINI D., SIOTTO E.: A high-level tool for curators of 3d virtual visits and its application to a virtual exhibition of renaissance frescoes. In *Proc. VAST 2010* (Paris, 2010), Eurographics/Blackwell Publishing, pp. 147–154. See page 188.

CLCG06. CORNELIS N., LEIBE B., CORNELIS K., GOOL L. V.: 3D City Modeling Using Cognitive Loops. In *3DPVT '06: Proceedings of the Third Intern. Symposium on 3D Data Processing, Visualization, and Transmission (3DPVT'06)* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 9–16. doi:10.1109/3DPVT.2006.1. See page 33.

Coq90. COQUILLART S.: Extended free-form deformation: a sculpturing tool for 3d geometric modeling. *SIGGRAPH Comput. Graph. 24*, 4 (Sept. 1990), 187–196. doi:10.1145/97880.97900. See pages 39 and 75.

CPCS06. CALLIERI M., PONCHIO F., CIGNONI P., SCOPIGNO R.: Easy access to huge 3d models of works of art. In *Fourth Eurographics Italian Chapter 2006* (2006), Fellner D., (Ed.), Eurographics Association, pp. 29–36. Catania, 22-24 Feb. 2006. URL: http://vcg.isti.cnr.it/Publications/2006/CPCS06. See page 188.

CPCS08. CALLIERI M., PONCHIO F., CIGNONI P., SCOPIGNO R.: Virtual inspector: a flexible visualizer for dense 3d scanned models. *IEEE Computer Graphics and Applications 28*, 1 (Jan.-Febr. 2008), 44–55. URL: http://vcg.isti.cnr.it/Publications/2008/CPCS08. See page 188.

CSR*08. COSTES E., SMITH C., RENTON M., GUÉDON Y., PRUSINKIEWICZ P., GODIN C.: MAppleT: simulation of apple tree development using mixed stochastic and biomechanical models. *Functional Plant Biology 35*, 10 (2008), 936+. doi:10.1071/fp08081. See page 37.

dBvKOS00. DE BERG M., VAN KREVELD M., OVERMARS M., SCHWARZKOPF O.: *Computational Geometry - Algorithms and Applications*, second ed. Springer, 2000. See page 50.

DF81. DOWNING F., FLEMMING U.: The bungalows of Buffalo. *Environment and Planning B: Planning and Design 8*, 3 (1981), 269–293. URL: http://EconPapers.repec.org/RePEc:pio:envirb:v:8:y:1981:i:3:p:269-293, doi:10.1068/b080269. See page 24.

DHL*98. DEUSSEN O., HANRAHAN P., LINTERMANN B., MĚCH R., PHARR M., PRUSINKIEWICZ P.: Realistic modeling and rendering of plant ecosystems. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1998), SIGGRAPH '98, ACM, pp. 275–286. doi:10.1145/280814.280898. See page 24.

Die05. DIESTEL R.: *Graph Theory*, third ed., vol. 173 of *Graduate Texts in Mathematics*. Springer-Verlag, Heidelberg, 2005. URL: http://diestel-graph-theory.com/. See page 46.

DJ08. DAVIES N., JOKINIEMI E.: *Dictionary of Architecture and Building Construction*. Architectural Press, 2008. See page 144.

DKT98. DEROSE T., KASS M., TRUONG T.: Subdivision surfaces in character animation. In *Proceedings of the 25th annual conference on Computer graphics and interactive* *techniques* (New York, NY, USA, 1998), SIGGRAPH '98, ACM, pp. 85–94. `doi:10.1145/280814.280826`. See page 60.

DL97. DEUSSEN O., LINTERMANN B.: A modelling method and user interface for creating plants. In *In Proceedings of Graphics Interface 97* (1997), Morgan Kaufmann Publishers, pp. 189–197. See page 36.

Dor10. DORMANS J.: Adventures in level design: generating missions and spaces for action adventure games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games* (New York, NY, USA, 2010), PCGames '10, ACM, pp. 1:1–1:8. `doi:10.1145/1814256.1814257`. See page 29.

dREF*88. DE REFFYE P., EDELIN C., FRANÇON J., JAEGER M., PUECH C.: Plant models faithful to botanical structure and development. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1988), SIGGRAPH '88, ACM, pp. 151–158. `doi:10.1145/54852.378505`. See page 37.

DRS07. DUARTE J. P., ROCHA J. A. M., SOARES G. D.: Unveiling the structure of the Marrakech Medina: A shape grammar and an interpreter for generating urban form. *Artif. Intell. Eng. Des. Anal. Manuf. 21*, 4 (Oct. 2007), 317–349. `doi:10.1017/S0890060407000315`. See page 24.

DS13. DASSAULT SYSTEMES: CATIA (Computer Aided Three-dimensional Interactive Application), 2013. [Accessed: 2013-11-27]. URL: `http://www.3ds.com/products-services/catia/`. See page 15.

DTT*10. DOERR M., TZOMPANAKI K., THEODORIDOU M., GEORGIS C., AXARIDOU A., HAVEMANN S.: A Repository for 3D Model Production and Interpretation in Culture and Beyond. In *The 8th EUROGRAPHICS Workshop on Graphics and Cultural Heritage, VAST10: The 11th International Symposium on Virtual Reality, Archaeology and Intelligent Cultural Heritage* (2010), pp. 97–104. `doi:10.2312/VAST/VAST10/097-104`. See pages 187 and 192.

Dua02. DUARTE J. P.: *Customizing mass housing : a discursive grammar for Siza's Malagueira houses*. PhD thesis, Massachusetts Institute of Technology. Dept. of Architecture, 2002. URL: `http://hdl.handle.net/1721.1/8189`. See page 24.

Dua05. DUARTE J. P.: Towards the mass customization of housing: the grammar of Siza's houses at Malagueira. *Environment and Planning B: Planning and Design 32*, 3 (2005), 347–380. URL: `http://EconPapers.repec.org/RePEc:pio:envirb:v:32:y:2005:i:3:p:347-380`, `doi:10.1068/b31124`. See page 24.

Ebe06. EBERLY D. H.: *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*, second ed. Morgan Kaufmann Publishers Inc., 2006. See page 64.

EE98. EPPSTEIN D., ERICKSON J.: Raising roofs, crashing cycles, and playing pool: applications of a data structure for finding pairwise interactions. In *Proceedings of the fourteenth annual symposium on Computational geometry* (New York, NY, USA, 1998), SCG '98, ACM, pp. 58–67. URL: `http://doi.acm.org/10.1145/276884.276891`, `doi:10.1145/276884.276891`. See pages 35 and 116.

EF01. EFROS A. A., FREEMAN W. T.: Image quilting for texture synthesis and transfer. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2001), SIGGRAPH '01, ACM, pp. 341–346. `doi:10.1145/383259.383296`. See page 37.

Ege13. EGER M.: *Locally Context-Sensitive Shape Grammars*. Master's thesis, TU Graz, Austria, 2013. See pages 177 and 178.

Esr13. ESRI: CityEngine, 2013. [Accessed: 2013-11-27]. URL: www.esri.com/software/cityengine. See pages 15, 26, 35, 138, and 191.

Eul67. EULER L.: Recherches sur la courbure des surfaces. *Memoires de l'academie des sciences de Berlin 16* (1767), 119–143. See page 51.

Far97. FARIN G.: *Curves and surfaces for computer-aided geometric design - a practical guide (4. ed.)*. Computer science and scientific computing. Academic Press, 1997. See page 76.

Fel92. FELLNER D. W.: *Computer Grafik*, second ed., vol. 58 of *Reihe Informatik*. B.I. Wissenschaftsverlag, 1992. See page 3.

Fle87. FLEMMING U.: More than the sum of parts: the grammar of Queen Anne houses. *Environment and Planning B: Planning and Design 14*, 3 (May 1987), 323–350. URL: http://ideas.repec.org/a/pio/envirb/v14y1987i3p323-350.html, doi:10.1068/b140323. See page 24.

GCP90. GURSOZ E. L., CHOI Y., PRINZ F. B.: Vertex-based representation of non-manifold boundaries. In *MIT-JSME Workshops* (1990). See page 71.

GE10. GRASL T., ECONOMOU A.: Palladian Graphs: Using a graph grammar to automate the Palladian grammar. In *28th eCAADe Conference Proceedings* (2010), eCAADe, pp. 275–283. See page 30.

GE11. GRASL T., ECONOMOU A.: GRAPE: using graph grammars to implement shape grammars. In *Proceedings of the 2011 Symposium on Simulation for Architecture and Urban Design* (San Diego, CA, USA, 2011), SimAUD '11, Society for Computer Simulation International, pp. 21–28. URL: http://dl.acm.org/citation.cfm?id=2048536.2048539. See page 30.

GHH*03. GRANADOS M., HACHENBERGER P., HERT S., KETTNER L., MEHLHORN K., SEEL M.: Boolean operations on 3d selective nef complexes data structure, algorithms, and implementation. In *Algorithms - ESA 2003: 11th Annual European Symposium* (Budapest, Hungary, September 2003), Di Battista G., Zwick U., (Eds.), vol. 2832 of *Lecture Notes in Computer Science*, Springer, pp. 174–186. URL: http://www.mpi-sb.mpg.de/~kettner/pub/nef_3d_esa_03_a.html. See page 71.

GMTF89. GOLDFEATHER J., MONAR S., TURK G., FUCHS H.: Near real-time csg rendering using tree normalization and geometric pruning. *IEEE Comput. Graph. Appl. 9*, 3 (May 1989), 20–28. doi:10.1109/38.28107. See pages 72 and 73.

Goo13a. GOOGLE: Google Earth, 2013. [Accessed: 2013-11-27]. URL: http://www.google.com/earth. See page 33.

Goo13b. GOOGLE: Google Maps, 2013. [Accessed: 2013-11-27]. URL: http://maps.google.com. See page 2.

Goo13c. GOOGLE: SketchUp, 2013. [Accessed: 2013-11-27]. URL: http://sketchup.google.com. See pages 15 and 138.

Goo13d. GOOGLE: Trimble 3D Warehouse, 2013. [Accessed: 2013-11-27]. URL: http://sketchup.google.com/3dwarehouse/. See pages 5 and 198.

GP92. GREEN T., PETRE M.: When visual programs are harder to read than textual programs. In *Proceedings of ECCE-6* (1992), pp. 167–180. URL: http://citeseerx.

ist.psu.edu/viewdoc/summary?doi=10.1.1.57.1633. See pages 17 and 118.

Gre89. GREENE N.: Voxel space automata: modeling with stochastic growth processes in voxel space. *SIGGRAPH Comput. Graph. 23*, 3 (July 1989), 175–184. doi:10.1145/74334.74351. See page 37.

GSMCO09. GAL R., SORKINE O., MITRA N. J., COHEN-OR D.: iwires: An analyze-and-edit approach to shape manipulation. *ACM Trans. Graph. 28*, 3 (July 2009), 33:1–33:10. doi:10.1145/1531326.1531339. See page 39.

GT06. GIERVELD J. D. J., TILBURG T. V.: A 6-item scale for overall, emotional, and social loneliness: Confirmatory tests on survey data. *Research on Aging 28*, 5 (2006), 582–598. URL: http://roa.sagepub.com/content/28/5/582.full.pdf.html, doi:10.1177/0164027506289723. See page 200.

Hav05. HAVEMANN S.: *Generative Mesh Modeling*. PhD thesis, Institute of Computer Graphics, Faculty of Computer Science, Braunschweig Technical University, Germany, November 2005. URL: www.digibib.tu-bs.de/?docid=00000008. See pages 14, 16, 45, 47, 55, 57, 80, 84, and 85.

Hec12. HECHER M.: *Grimaldo - A Scriptable Framework for Developing Visual Applications*. Master's thesis, Insitute of Computer Graphics and Knowlegde Visualization, Graz University of Technology, Austria, 2012. See pages 65, 67, 68, and 89.

HF04. HAVEMANN S., FELLNER D.: Generative parametric design of gothic window tracery. In *Shape Modeling Applications, 2004. Proceedings* (june 2004), pp. 350–353. doi:10.1109/SMI.2004.1314525. See page 85.

HHKF10. HOHMANN B., HAVEMANN S., KRISPEL U., FELLNER D.: A GML shape grammar for semantically enriched 3D building models. *Computers & Graphics 34*, 4 (2010), 322–334. Procedural Methods in Computer Graphics; Illustrative Visualization. doi:DOI:10.1016/j.cag.2010.05.007. See pages 28, 91, and 95.

HJA02. HOFFMANN C. M., JOAN-ARINYO R.: Parametric modeling. In *Handbook of Computer Aided Geometric Design*. Elsevier, 2002, ch. 21, pp. 519–542. See page 17.

HJO*01. HERTZMANN A., JACOBS C. E., OLIVER N., CURLESS B., SALESIN D. H.: Image analogies. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2001), SIGGRAPH '01, ACM, pp. 327–340. doi:10.1145/383259.383295. See page 37.

HMDB09. HORNA S., MENEVEAUX D., DAMIAND G., BERTRAND Y.: Consistency constraints and 3d building reconstruction. *Computer-Aided Design (CAD) 41*, 1 (January 2009), 13–27. doi:10.1016/j.cad.2008.11.006. See page 35.

HME*12. HECHER M., MÖSTL R., EGGELING E., DERLER C., FELLNER D. W.: 'Tangible Culture' – Designing Virtual Exhibitions on Multi-Touch Devices. In *Social Shaping of Digital Publishing, ELPUB 2012* (2012), Baptista A. A., Linde P., Lavesson N., de Brito M. A., (Eds.), IOS Press, pp. 104–113. See page 188.

Hon71. HONDA H.: Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the shape of the tree-like body. *Journal of Theoretical Biology 31*, 2 (1971), 331–338. URL: http://www.sciencedirect.com/science/article/pii/0022519371901913, doi:10.1016/0022-5193(71)90191-3. See page 36.

Hop96. HOPPE H.: Progressive meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1996), SIGGRAPH '96, ACM, pp. 99–108. doi:10.1145/237170.237216. See page 55.

HRJS98. HEMPHILL S. T., REINITZ I. M., JOHNSON M. L., SHIGLEY J. E.: Modeling the appearance of the round brilliant cut diamond: An analysis of brilliance. *Gems & Gemology 34* (1998), 158–183. See page 162.

HT13. HUANG C.-Y., TAI W.-K.: Ting tools: interactive and procedural modeling of chinese ting. *The Visual Computer 29*, 12 (2013), 1303–1318. URL: http://dx.doi.org/10.1007/s00371-012-0771-3, doi:10.1007/s00371-012-0771-3. See pages 35 and 184.

Ins13. INSTITUTE OF COMPUTER GRAPHICS AND KNOWLEDGE VISUALIZATION: GML Wiki, 2013. [Accessed: 2013-11-27]. URL: https://hydra.cgv.tugraz.at/gmlwiki/index.php. See pages 80 and 83.

JK13. JOHANNKAISER, Ringmanufaktur, 2013. [Accessed: 2013-12-04]. URL: http://www.mytrauring.de. See pages 158 and 159.

KAC06. KAHL W., ANAND C. K., CARETTE J.: Control-flow semantics for assembly-level data-flow graphs. In *Proceedings of the 8th international conference on Relational Methods in Computer Science, Proceedings of the 3rd international conference on Applications of Kleene Algebra* (Berlin, Heidelberg, 2006), RelMiCS'05, Springer-Verlag, pp. 147–160. doi:10.1007/11734673_12. See page 120.

KCKK12. KALOGERAKIS E., CHAUDHURI S., KOLLER D., KOLTUN V.: A probabilistic model for component-based shape synthesis. *ACM Trans. Graph. 31*, 4 (July 2012), 55:1–55:11. doi:10.1145/2185520.2185551. See pages 39 and 40.

KE81. KONING H., EIZENBERG J.: The language of the prairie: Frank Lloyd Wright's prairie houses. *Environment and Planning B: Planning and Design 8*, 3 (1981), 295–323. URL: http://EconPapers.repec.org/RePEc:pio:envirb:v:8:y:1981:i:3:p:295-323, doi:10.1068/b080295. See page 24.

KHF10. KRISPEL U., HAVEMANN S., FELLNER D. W.: Famos - a visual editor for hierachical volumetric modeling. In *Tagungsband 05. Kongress Multimediatechnik Wismar* (2010). See page 29.

KK11. KRECKLAU L., KOBBELT L.: Procedural modeling of interconnected structures. *Computer Graphics Forum 30*, 2 (2011), 335–344. doi:10.1111/j.1467-8659.2011.01864.x. See pages 27, 28, 91, 95, 102, 115, and 132.

KK12. KRECKLAU L., KOBBELT L.: Interactive modeling by procedural high-level primitives. *Computers & Graphics 36*, 5 (2012), 376–386. Shape Modeling International (SMI) Conference 2012. URL: http://www.sciencedirect.com/science/article/pii/S0097849312000672, doi:10.1016/j.cag.2012.03.028. See page 29.

Kni80. KNIGHT T. W.: The generation of hepplewhite-style chair-back designs. *Environment and Planning B: Planning and Design 7*, 2 (1980), 227–238. URL: http://EconPapers.repec.org/RePEc:pio:envirb:v:7:y:1980:i:2:p:227-238, doi:10.1068/b070227. See page 24.

KPK10. KRECKLAU L., PAVIC D., KOBBELT L.: Generalized use of non-terminal symbols for procedural modeling. *Computer Graphics Forum 29*, 8 (2010), 2291–2303. doi:10.1111/j.1467-8659.2010.01714.x. See pages 16, 21, 27, and 171.

Kriar. KRISPEL U.: *Procedural Convex Complexes for Robust Evaluation of Generative Architecture (working title).* PhD thesis, Insitute of Computer Graphics and Knowlegde Visualization, Graz University of Technology, Austria, (to appear). See pages 8, 63, 64, 70, 85, 99, 100, 105, and 206.

KST*09. KOUTSOURAKIS P., SIMON L., TEBOUL O., TZIRITAS G., PARAGIOS N.: Single view reconstruction using shape grammars for urban environments. In *ICCV09* (2009). See page 33.

KUF14. KRISPEL U., ULLRICH T., FELLNER D. W.: Fast and exact plane-based representation for polygonal meshes. In *Proceedings of the 8th International Conference on Computer Graphics, Visualization, Computer Vision and Image Processing 2014,* (2014). See page 63.

KW11. KELLY T., WONKA P.: Interactive architectural modeling with procedural extrusions. *ACM Trans. Graph. 30*, 2 (Apr. 2011), 14:1–14:15. doi:10.1145/1944846.1944854. See pages 35 and 36.

LD99. LINTERMANN B., DEUSSEN O.: Interactive modeling of plants. *Computer Graphics and Applications, IEEE 19*, 1 (jan/feb 1999), 56–65. doi:10.1109/38.736469. See page 36.

LDG01. LEGAKIS J., DORSEY J., GORTLER S.: Feature-based cellular texturing for architectural models. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), SIGGRAPH '01, ACM, pp. 309–316. doi:10.1145/383259.383293. See page 38.

LG06. LARIVE M., GAILDRAT V.: Wall grammar for building generation. In *Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia* (New York, NY, USA, 2006), GRAPHITE '06, ACM, pp. 429–437. doi:10.1145/1174429.1174501. See page 28.

LHP11. LEBLANC L., HOULE J., POULIN P.: Component-based modeling of complete buildings. In *Proceedings of Graphics Interface 2011* (School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2011), GI '11, Canadian Human-Computer Communications Society, pp. 87–94. URL: http://dl.acm.org/citation.cfm?id=1992917.1992932. See page 28.

Lin68. LINDENMAYER A.: Mathematical models for cellular interactions in development II. simple and branching filaments with two-sided inputs. *Journal of Theoretical Biology 18*, 3 (1968), 300–315. URL: http://www.sciencedirect.com/science/article/pii/0022519368900805, doi:10.1016/0022-5193(68)90080-5. See pages 19 and 20.

LLD12. LASRAM A., LEFEBVRE S., DAMEZ C.: Procedural texture preview. *Comp. Graph. Forum 31*, 2pt2 (May 2012), 413–420. doi:10.1111/j.1467-8659.2012.03020.x. See page 38.

LOMI11. LAU M., OHGAWARA A., MITANI J., IGARASHI T.: Converting 3D furniture models to fabricatable parts and connectors. *ACM Trans. Graph. 30*, 4 (July 2011), 85:1–85:6. doi:10.1145/2010324.1964980. See page 32.

LWW08. LIPP M., WONKA P., WIMMER M.: Interactive visual editing of grammars for procedural architecture. *ACM Transactions on Graphics 27*, 3 (Aug. 2008), 102:1–10. Article No. 102. URL: http://www.cg.tuwien.ac.at/research/publications/2008/LIPP-2008-IEV/. See page 29.

LZS*11. LI Y., ZHENG Q., SHARF A., COHEN-OR D., CHEN B., MITRA N. J.: 2D-3D fusion for layer decomposition of urban facades. In *IEEE International Conference on Computer Vision (ICCV)* (Barcelona, Spain, 11 2011). See page 34.

Mai02. MAIERHOFER S.: *Rule-Based Mesh Growing and Generalized Subdivision Meshes*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, 2002. URL: http://www.cg.tuwien.ac.at/research/publications/2002/Maierhofer-thesis/. See page 21.

Man82. MANDELBROT B. B.: *The fractal geometry of nature*, 1 ed. W.H. Freeman, Aug. 1982. URL: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0716711869. See page 19.

Män88. MÄNTYLÄ M.: *An Introduction to Solid Modeling*. Principles of computer science series. Computer Science Press, 1988. URL: http://books.google.ca/books?id=N7BEPgAACAAJ. See pages 50 and 68.

Mar13. MARIUS KINTEL AND CLIFFORD WOLF: OpenSCAD, 2013. [Accessed: 2013-11-27]. URL: http://http://www.openscad.org/. See page 16.

MDDR13. MURRAY S., DUNCAN G., DOYLE R., REAM D.: Hello Games, 2013. [Accessed: 2013-12-13]. URL: http://www.hellogames.org/. See page 15.

MDH*10. MENZ S., DAMMERTZ H., HANIKA J., WEBER M., LENSCH H. P. A.: Graphical interface models for procedural mesh growing. In *VMV* (2010), pp. 17–24. doi:10.2312/PE/VMV/VMV10/017-024. See page 21.

Mic13a. MICROSOFT: Bing Maps Platform, 2013. [Accessed: 2013-11-27]. URL: http://www.microsoft.com/maps/. See page 33.

Mic13b. MICROSOFT: Kinect Sensor, 2013. [Accessed: 2013-11-27]. URL: http://www.microsoft.com/en-us/kinectforwindows/. See pages 4, 40, 193, 197, and 198.

Mic13c. MICROSOFT: Office, 2013. [Accessed: 2013-11-27]. URL: http://office.microsoft.com/. See page 2.

Mit92. MITCHELL W.-J.: *The Logic of Architecture*. The MIT Press, 1992. See page 144.

MMPD08. MURG S., MORITSCH O., PENSOLD W., DERLER C.: Using standardized methods to present three-dimensional content on the web in the context of cultural heritage. In *Museums and the Web 2008* (Toronto, March 2008), Trant J., Bearman D., (Eds.), Archives & Museum Informatics. See page 188.

MMWVG11. MATHIAS M., MARTINOVIC A., WEISSENBERG J., VAN GOOL L.: Procedural 3D building reconstruction using shape grammars and detectors. In *3D Imaging, Modeling, Processing, Visualization and Transmission (3DIMPVT), 2011 International Conference on* (may 2011), pp. 304–311. doi:10.1109/3DIMPVT.2011.45. See page 33.

MP96. MĚCH R., PRUSINKIEWICZ P.: Visual models of plants interacting with their environment. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1996), SIGGRAPH '96, ACM, pp. 397–410. doi:10.1145/237170.237279. See pages 21 and 37.

MP02. MARCHEIX D., PIERRA G.: A survey of the persistent naming problem. In *Proceedings of the Seventh ACM Symposium on Solid Modeling and Applications* (New York, NY, USA, 2002), SMA '02, ACM, pp. 13–22. doi:10.1145/566282.566288. See page 17.

MSK10. MERRELL P., SCHKUFZA E., KOLTUN V.: Computer-generated residential building layouts. *ACM Trans. Graph. 29*, 6 (Dec. 2010), 181:1–181:12. doi:10.1145/1882261.1866203. See page 35.

MSLV08. MATEEVITSI V., SFAKIANOS M., LEPOURAS G., VASSILAKIS C.: A game-engine based virtual museum authoring and presentation system. In *Proc. Digital Interactive Media in Entertainment and Arts* (New York, NY, USA, 2008), DIMEA '08, ACM, pp. 451–457. doi:10.1145/1413634.1413714. See page 188.

MVW*06. MÜLLER P., VEREENOOGHE T., WONKA P., PAAP I., VAN GOOL L.: Procedural 3D reconstruction of Puuc buildings in Xkipché. In *Proceedings of the 7th International conference on Virtual Reality, Archaeology and Intelligent Cultural Heritage* (Aire-la-Ville, Switzerland, Switzerland, 2006), VAST'06, Eurographics Association, pp. 139–146. doi:10.2312/VAST/VAST06/139-146. See page 27.

MWA*12. MUSIALSKI P., WONKA P., ALIAGA D. G., WIMMER M., VAN GOOL L., PURGATHOFER W.: A Survey of Urban Reconstruction. In *EUROGRAPHICS 2012 State of the Art Reports* (2012), Eurographics Association, pp. 1–28. URL: http://www.cg.tuwien.ac.at/research/publications/2012/musialski-2012-sur/, doi:10.2312/conf/EG2012/stars/001-028. See page 33.

MWA*13. MUSIALSKI P., WONKA P., ALIAGA D. G., WIMMER M., VAN GOOL L., PURGATHOFER W.: A Survey of Urban Reconstruction. *Computer Graphics Forum 32*, 6 (2013), 146–177. doi:10.1111/cgf.12077. See page 33.

MWH*06. MÜLLER P., WONKA P., HAEGLER S., ULMER A., VAN GOOL L.: Procedural modeling of buildings. *ACM Transactions on Graphics 25*, 3 (July 2006), 614–623. doi:10.1145/1141911.1141931. See pages 16, 26, 27, 91, 94, 95, 98, 101, 181, 207, and 208.

MWW12. MUSIALSKI P., WIMMER M., WONKA P.: Interactive coherence-based façade modeling. *Comp. Graph. Forum 31*, 2pt3 (May 2012), 661–670. doi:10.1111/j.1467-8659.2012.03045.x. See page 29.

MWZ*13. MITRA N. J., WAND M., ZHANG H., COHEN-OR D., BOKELOH M.: Structure-aware shape processing. In *EUROGRAPHICS State-of-the-art Report* (2013). See pages 38 and 40.

MZWVG07. MÜLLER P., ZENG G., WONKA P., VAN GOOL L.: Image-based procedural modeling of facades. *ACM Transactions on Graphics 26*, 3 (July 2007), 85:1–85:9. doi:10.1145/1276377.1276484. See page 33.

Nef78. NEF W.: *Beiträge zur Theorie der Polyeder.* Beiträge zur Mathematik, Informatik und Nachrichtentechnik. Herbert Lang, Bern, 1978. See page 71.

Opp86. OPPENHEIMER P. E.: Real time design and animation of fractal plants and trees. *SIGGRAPH Comput. Graph. 20*, 4 (Aug. 1986), 55–64. doi:10.1145/15886.15892. See page 36.

OS93. ORTWEIN A., SCHEFFERS A.: *Deutsche Renaissance,*, second ed. Seemann Verlag, 1893. See page 144.

Pal07. PALUBICKI W.: *Fuzzy Plant Modeling with OpenGL- Novel Approaches in Simulating Phototropism and Environmental Conditions*. VDM Verlag, Saarbrücken, Germany, Germany, 2007. See page 37.

Pat12. PATOW G.: User-friendly graph editing for procedural modeling of buildings. *IEEE Comput. Graph. Appl. 32*, 2 (Mar. 2012), 66–75. doi:10.1109/MCG.2010.104. See pages 29 and 118.

PBH*10. PAN X., BECKMANN P., HAVEMANN S., TZOMPANAKI K., DOERR M., FELLNER D. W.: A Distributed Object Repository for Cultural Heritage. In *The 8th EUROGRAPHICS Workshop on Graphics and Cultural Heritage, VAST10: The 11th International Symposium on Virtual Reality, Archaeology and Intelligent Cultural Heritage* (2010), pp. 105–114. doi:10.2312/VAST/VAST10/105-114. See page 187.

PC02. PUGLIESE M., CAGAN J.: Capturing a rebel: modeling the Harley-Davidson brand through a motorcycle shape grammar. *Research in Engineering Design 13* (2002), 139–156. doi:10.1007/s00163-002-0013-1. See page 24.

PHHM97. PRUSINKIEWICZ P., HAMMEL M., HANAN J., MĚCH R.: Visual models of plant development. In *Handbook of formal languages, vol. 3*, Rozenberg G., Salomaa A., (Eds.). Springer-Verlag New York, Inc., New York, NY, USA, 1997, pp. 535–597. URL: http://dl.acm.org/citation.cfm?id=267871.267880. See page 20.

PHL*09. PALUBICKI W., HOREL K., LONGAY S., RUNIONS A., LANE B., MĚCH R., PRUSINKIEWICZ P.: Self-organizing tree models for image synthesis. *ACM Trans. Graph. 28*, 3 (July 2009), 58:1–58:10. doi:10.1145/1531326.1531364. See page 37.

PHM00. PRUSINKIEWICZ P., HANAN J., MECH R.: An L-System-Based Plant Modeling Language. In *Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance* (London, UK, UK, 2000), AGTIVE ’99, Springer-Verlag, pp. 395–410. URL: http://dl.acm.org/citation.cfm?id=646676.702142. See page 21.

PJM94. PRUSINKIEWICZ P., JAMES M., MĚCH R.: Synthetic topiary. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1994), SIGGRAPH ’94, ACM, pp. 351–358. doi:10.1145/192161.192254. See pages 20 and 37.

PKMH00. PRUSINKIEWICZ P., KARWOWSKI R., MECH R., HANAN J.: L-studio/cpfg: A software system for modeling plants. In *Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance* (London, UK, UK, 2000), AGTIVE ’99, Springer-Verlag, pp. 457–464. URL: http://dl.acm.org/citation.cfm?id=646676.702139. See page 21.

PL90. PRUSINKIEWICZ P., LINDENMAYER A.: *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990. See page 20.

PM01. PARISH Y. I. H., MÜLLER P.: Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), SIGGRAPH ’01, ACM, pp. 301–308. doi:10.1145/383259.383292. See page 21.

PMKL01. PRUSINKIEWICZ P., MÜNDERMANN L., KARWOWSKI R., LANE B.: The use of positional information in the modeling of plants. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), SIGGRAPH ’01, ACM, pp. 289–300. doi:10.1145/383259.383291. See pages 21 and 36.

Pos13. POSCH M.: *Window Modeling With Procedural Building Blocks*. Tech. rep., Institute of Computer Graphics and Knowledge Visualization, Graz University of Technology, 2013. See pages 142 and 147.

Pru86. PRUSINKIEWICZ P.: Graphical applications of L-systems. In *Proceedings on Graphics Interface '86/Vision Interface '86* (Toronto, Ont., Canada, Canada, 1986), Canadian Information Processing Society, pp. 247–253. URL: `http://dl.acm.org/citation.cfm?id=16564.16608`. See page 19.

PSK*12. PIRK S., STAVA O., KRATT J., SAID M. A. M., NEUBERT B., MĚCH R., BENEŠ B., DEUSSEN O.: Plastic trees: interactive self-adapting botanical tree models. *ACM Trans. Graph. 31*, 4 (July 2012), 50:1–50:10. `doi:10.1145/2185520.2185546`. See page 37.

Psz14. PSZEIDA M.: *A Partial Reconstruction of Louvre Facades*. Tech. rep., Institute of Computer Graphics and Knowledge Visualization, Graz University of Technology, 2014. See pages 138, 162, and 165.

RA99. RAMAMOORTHI R., ARVO J.: Creating generative models from range images. In *In Proceedings of SIGGRAPH 99* (1999). See page 30.

RB85. REEVES W. T., BLAU R.: Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1985), SIGGRAPH '85, ACM, pp. 313–322. `doi:10.1145/325334.325250`. See page 36.

RB07. RIPPERDA N., BRENNER C.: Data driven rule proposal for grammar based facade reconstruction. In *PIA07* (2007), pp. 1–6. See page 33.

RB09. RIPPERDA N., BERENNER C.: Application of a formal grammar to facade reconstruction in semiautomatic and automatic environments. In *AGILE09* (2009). See page 33.

Rip08a. RIPPERDA N.: Determination of facade attributes for facade reconstruction. In *Intern. Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences* (2008), pp. 285–290. See page 33.

Rip08b. RIPPERDA N.: Grammar based facade reconstruction using rjmcmc. In *Photogrammetrie Fernerkundung Geoinformation (PFG)* (2008), pp. 83–92. See page 33.

RKT*12. RIEMENSCHNEIDER H., KRISPEL U., THALLER W., DONOSER M., HAVEMANN S., FELLNER D., BISCHOF H.: Irregular lattices for complex shape grammar facade parsing. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on* (june 2012), pp. 1640–1647. `doi:10.1109/CVPR.2012.6247857`. See pages 33 and 34.

RLP07. RUNIONS A., LANE B., PRUSINKIEWICZ P.: Modeling trees with a space colonization algorithm. In *Proceedings of the Third Eurographics conference on Natural Phenomena* (Aire-la-Ville, Switzerland, Switzerland, 2007), NPH'07, Eurographics Association, pp. 63–70. `doi:10.2312/NPH/NPH07/063-070`. See page 21.

RO89. ROSSIGNAC J., O'CONNOR M.: *SGC: A Dimension-independent Model for Pointsets with Internal Structures and Incomplete Boundaries*. Research report. IBM T.J. Watson Research Center, 1989. URL: `http://books.google.ca/books?id=a76UtgAACAAJ`. See page 71.

Rob13a. ROBERT MCNEEL & ASSOCIATES: Grasshopper 3D, 2013. [Accessed: 2013-11-27]. URL: `http://www.grasshopper3d.com/`. See pages 17, 118, and 125.

Rob13b. ROBERT MCNEEL & ASSOCIATES: Rhinoceros 3D, 2013. [Accessed: 2013-11-27]. URL: http://www.rhino3d.com. See page 17.

RVN13. REINERS D., VOSS G., NEUMANN C.: OpenSG, 2013. [Accessed: 2013-11-27]. URL: http://www.opensg.org/. See pages 67 and 89.

Sch08. SCHULZE J.: Wie man um 1874 Fenster baute. In *Fenster im Baudenkmal* (2008), Lukas Verlag. See page 144.

Sch14. SCHWARZ M.: *Semantically Consistent Tavern in GML.* Tech. rep., Institute of Computer Graphics and Knowledge Visualization, Graz University of Technology, 2014. See page 186.

SFCH12. SHEN C.-H., FU H., CHEN K., HU S.-M.: Structure recovery by part assembly. *ACM Trans. Graph. 31*, 6 (Nov. 2012), 180:1–180:11. doi:10.1145/2366145.2366199. See page 40.

SG72. STINY G., GIPS J.: Shape Grammars and the Generative Specification of Painting and Sculpture. In *Information Processing '71* (Amsterdam, 1972), Friedman C. V., (Ed.), International Federation for Information Processing, North Holland Publishing Co., pp. 1460–1465. See pages 22 and 206.

SI90. SUGIHARA K., IRI M.: A solid modelling system free from topological inconsistency. *J. Inf. Process. 12*, 4 (Apr. 1990), 380–393. URL: http://dl.acm.org/citation.cfm?id=81617.81622. See page 63.

Sid13. SIDE EFFECTS SOFTWARE: Houdini, 2013. [Accessed: 2013-11-27]. URL: http://www.sidefx.com. See pages 17, 118, and 125.

SK92. SNYDER J. M., KAJIYA J. T.: Generative modeling: A symbolic system for geometric modeling. *Computer Graphics 26* (1992), 369–378. See page 14.

SLJ98. STEWART N., LEACH G., JOHN S.: An improved z-buffer csg rendering algorithm. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (New York, NY, USA, 1998), HWWS '98, ACM, pp. 25–30. doi:10.1145/285305.285308. See page 73.

SM78. STINY G., MITCHELL W. J.: The Palladian grammar. *Environment and Planning B: Planning and Design 5*, 1 (January 1978), 5–18. URL: http://ideas.repec.org/a/pio/envirb/v5y1978i1p5-18.html, doi:10.1068/b050005. See pages 24 and 30.

Smi84. SMITH A. R.: Plants, fractals, and formal languages. *SIGGRAPH Comput. Graph. 18*, 3 (Jan. 1984), 1–10. doi:10.1145/964965.808571. See page 20.

SP86. SEDERBERG T. W., PARRY S. R.: Free-form deformation of solid geometric models. *SIGGRAPH Comput. Graph. 20*, 4 (Aug. 1986), 151–160. doi:10.1145/15886.15903. See pages 39 and 75.

Sta98. STAM J.: Exact evaluation of catmull-clark subdivision surfaces at arbitrary parameter values. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1998), SIGGRAPH '98, ACM, pp. 395–404. doi:10.1145/280814.280945. See page 60.

Ste01. STEVENS N.: Combating loneliness: a friendship enrichment programme for older women. *Ageing and Society 21* (3 2001), 183–202. URL: http://journals.cambridge.org/article_S0144686X01008108, doi:10.1017/S0144686X01008108. See page 197.

Sti77. STINY G.: Ice-ray: a note on the generation of chinese lattice designs. *Environment and Planning B: Planning and Design 4*, 1 (1977), 89–98. URL: http://EconPapers.repec.org/RePEc:pio:envirb:v:4:y:1977:i:1:p:89-98, doi:10.1068/b040089. See pages 22 and 23.

Sti80. STINY G.: Introduction to shape and shape grammars. *Environment and Planning B 7*, 3 (1980), 343–351. doi:10.1068/b070343. See pages 22 and 23.

Sti82. STINY G.: Spatial relations and grammars. *Environment and Planning B: Planning and Design 9*, 1 (1982), 113–114. URL: http://EconPapers.repec.org/RePEc:pio:envirb:v:9:y:1982:i:1:p:113-114. See page 24.

Teo09. TEOH S. T.: Generalized descriptions for the procedural modeling of ancient east asian buildings. In *Proceedings of the Fifth Eurographics conference on Computational Aesthetics in Graphics, Visualization and Imaging* (Aire-la-Ville, Switzerland, Switzerland, 2009), Computational Aesthetics'09, Eurographics Association, pp. 17–24. doi:10.2312/COMPAESTH/COMPAESTH09/017-024. See pages 34 and 184.

.th13a. .THEPRODUKKT: .kkrieger, 2013. [Accessed: 2013-11-27]. URL: http://www.farb-rausch.de/prod.py?which=114. See pages 15 and 16.

.th13b. .THEPRODUKKT: .werkkzeug, 2013. [Accessed: 2013-11-27]. URL: http://www.farb-rausch.de/prod.py?which=113. See page 15.

Thaar. THALLER W.: *The Language and Structure of Procedural Modeling (working title)*. PhD thesis, Insitute of Computer Graphics and Knowlegde Visualization, Graz University of Technology, Austria, (to appear). See pages 8, 118, 121, and 206.

TKH*11. THALLER W., KRISPEL U., HAVEMANN S., REDI I., REDI A., FELLNER D. W.: Developing parametric building models - the GANDIS use case. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences XXXVIII-5/W16* (2011), 163–170. URL: http://www.int-arch-photogramm-remote-sens-spatial-inf-sci.net/XXXVIII-5-W16/163/2011/, doi:10.5194/isprsarchives-XXXVIII-5-W16-163-2011. See page 28.

TKHF12. THALLER W., KRISPEL U., HAVEMANN S., FELLNER D.: Implicit nested repetition in dataflow for procedural modeling. In *COMPUTATION TOOLS 2012* (2012), Ullrich T., Lorenz P., (Eds.), IARIA, pp. 45–50. See pages 118, 121, 122, 123, and 132.

TKZ*13a. THALLER W., KRISPEL U., ZMUGG R., HAVEMANN S., FELLNER D. W.: A Graph-Based Language for Direct Manipulation of Procedural Models. *International Journal On Advances in Software 6*, 3 & 4 (12 2013), 255–236. See pages xi, 118, 121, 122, 125, 133, 134, and 135.

TKZ*13b. THALLER W., KRISPEL U., ZMUGG R., HAVEMANN S., FELLNER D. W.: Shape grammars on convex polyhedra. *Computers & Graphics 37*, 6 (3 2013), 707–717. Shape Modeling International (SMI) Conference 2013. URL: http://www.sciencedirect.com/science/article/pii/S0097849313000861, doi:10.1016/j.cag.2013.05.012. See pages xi, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 167, 170, 180, 181, and 182.

TLL*11. TALTON J. O., LOU Y., LESSER S., DUKE J., MĚCH R., KOLTUN V.: Metropolis procedural modeling. *ACM Trans. Graph. 30*, 2 (Apr. 2011), 11:1–11:14. doi:10.1145/1944846.1944851. See page 32.

TMT10. TOSHEV A., MORDOHAI P., TASKAR B.: Detecting and parsing architecture at city scale from range data. In *CVPR* (2010), IEEE, pp. 398–405. URL: http://dblp.

uni-trier.de/db/conf/cvpr/cvpr2010.html#ToshevMT10. See page 33.

TMW02. Tobler R. F., Maierhofer S., Wilkie A.: Mesh-based parametrized L-systems and generalized subdivision for generating complex geometry. *International Journal of Shape Modeling 8*, 2 (2002), 173–191. URL: http://www.cg.tuwien.ac.at/research/publications/2002/tobler02_mbpls/. See page 21.

Tor13. Torus Knot Software Ltd.: OGRE, 2013. [Accessed: 2013-11-27]. URL: http://www.ogre3d.org/. See page 188.

TPC*10. Tarini M., Pietroni N., Cignoni P., Panozzo D., Puppo E.: Practical quad mesh simplification. *Computer Graphics Forum 29*, 2 (2010), 407–418. doi:10.1111/j.1467-8659.2009.01610.x. See page 51.

TSKP10. Teboul O., Simon L., Koutsourakis P., Paragios N.: Segmentation of building facades using procedural shape priors. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on* (2010), pp. 3105–3112. doi:10.1109/CVPR.2010.5540068. See page 33.

TSP*12. Trapp M., Semmo A., Pokorski R., Herrmann C.-D., Döllner J., Eichhorn M., Heinzelmann M.: Colonia 3d - communication of virtual 3d reconstructions in public spaces. *International Journal of Heritage in the Digital Era (IJHDE) 1*, 1 (2012), 45–74. See page 188.

TZK*13. Thaller W., Zmugg R., Krispel U., Posch M., Havemann S., Fellner D. W.: Creating procedural window building blocks using the generative fact labeling method. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences XL-5/W1* (2013), 235–242. URL: http://www.int-arch-photogramm-remote-sens-spatial-inf-sci.net/XL-5-W1/235/2013/, doi:10.5194/isprsarchives-XL-5-W1-235-2013. See pages xi, 141, 145, 146, 147, 148, 149, 151, 152, 153, and 154.

UF07. Ullrich T., Fellner D. W.: Robust shape fitting and semantic enrichment. In *CIPA 2007 Conf. Proc.* (2007), pp. 727–732. See page 30.

UF11a. Ullrich T., Fellner D. W.: Generative object definition and semantic recognition. In *3D Object Retrieval 2011, Eurographics Symposium Proceedings* (2011), Laga H., Schreck T., Ferreira A., Godil A., Pratikakis I., Veltkamp R., (Eds.), Eurographics Association, pp. 1–8. See page 31.

UF11b. Ullrich T., Fellner D. W.: Linear algorithms in sublinear time – a tutorial on statistical estimation. *IEEE Computer Graphics and Applications 31*, 2 (2011), 58–66. doi:10.1109/MCG.2010.21. See page 31.

UIM12. Umetani N., Igarashi T., Mitra N. J.: Guided exploration of physically valid shapes for furniture design. *ACM Trans. Graph. 31*, 4 (July 2012), 86:1–86:11. doi:10.1145/2185520.2185582. See page 39.

Ula62. Ulam S.: On Some Mathematical Problems Connected with Patterns of Growth and Figures. In *American Mathematical Society Proceedings of Symposia in Applied Mathematics* (1962), vol. 14, American Mathematical Society, pp. 215–224. See page 36.

Uni13. Unity Technologies: Unity3D, 2013. [Accessed: 2013-11-27]. URL: http://unity3d.com/. See page 187.

USF08. Ullrich T., Settgast V., Fellner D. W.: Semantic fitting and reconstuction. *Journal on Computing and Cultural Heritage (JOCCH) 1* (2008), 1–20. doi:10.1145/1434763.1434769. See page 31.

USSF11. Ullrich T., Schinko C., Schiffer T., Fellner D. W.: Variance analysis and comparison in computer-aided design. In *Proceedings of the 4th ISPRS International Workshop 3D-ARCH 2011* (2011), p. 5. See page 31.

V2m13. V2me, Virtual Coach Reaches Out To Me, 2013. [Accessed: 2013-11-27]. URL: http://www.v2me.org/. See page 193.

VAB10. Vanegas C. A., Aliaga D. G., Benevs B.: Building reconstruction using manhattan-world grammars. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on 0* (2010), 358–365. doi:10.1109/CVPR.2010.5540190. See page 34.

VGZVdBM07. Van Gool L., Zeng G., Van den Borre F., Müller P.: Towards mass-produced building models. In *Photogrammetric Image Analysis* (September 2007), Stilla U., Mayer H., Rottensteiner F., Heipke C., Hinz S., (Eds.), Institute of Photogrammetry and Cartography, Technische Universitaet Muenchen, pp. 209–220. See page 33.

VKW*12. Vanegas C. A., Kelly T., Weber B., Halatsch J., Aliaga D. G., Müller P.: Procedural generation of parcels in urban modeling. *Comp. Graph. Forum 31*, 2pt3 (May 2012), 681–690. doi:10.1111/j.1467-8659.2012.03047.x. See page 35.

VSG13. VSG - Visualization Sciences Group: Open Inventor, 2013. [Accessed: 2013-11-27]. URL: http://www.vsg3d.com/open-inventor/sdk. See page 67.

WMD*04. White M., Mourkoussis N., Darcy J., Petridis P., Liarokapis F., Lister P., Walczak K., Wojciechowski K., Cellary W., Chmielewski J., Stawniak M., Wiza W., Patel M., Stevenson J., Manley J., Giorgini F., Sayd P., Gaspard F.: Arco – an architecture for digitization, management and presentation of virtual exhibitions. In *Proc. Computer Graphics International* (Washington, DC, USA, 2004), CGI '04, IEEE Computer Society, pp. 622–625. doi:10.1109/CGI.2004.16. See page 188.

WOD09. Whiting E., Ochsendorf J., Durand F.: Procedural modeling of structurally-sound masonry buildings. *ACM Trans. Graph. 28*, 5 (Dec. 2009), 112:1–112:9. doi:10.1145/1618452.1618458. See page 27.

WP95. Weber J., Penn J.: Creation and rendering of realistic trees. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1995), SIGGRAPH '95, ACM, pp. 119–128. doi:10.1145/218380.218427. See page 36.

WWSR03. Wonka P., Wimmer M., Sillion F., Ribarsky W.: Instant architecture. *ACM Transactions on Graphics 22*, 3 (July 2003), 669–677. doi:10.1145/882262.882324. See pages 24, 26, 27, 91, and 95.

WWWC04. Wojciechowski R., Walczak K., White M., Cellary W.: Building virtual and augmented reality museum exhibitions. In *Proceedings of the ninth international conference on 3D Web technology* (New York, NY, USA, 2004), Web3D '04, ACM, pp. 135–144. doi:10.1145/985040.985060. See page 188.

WZS98. Wong M. T., Zongker D. E., Salesin D. H.: Computer-generated floral ornament. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1998), SIGGRAPH '98, ACM, pp. 423–434. doi:10.1145/280814.280948. See page 24.

Xfr13. Xfrog Inc.: Xfrog, 2013. [Accessed: 2013-11-27]. URL: www.xfrog.com. See pages 15 and 36.

XZCOC12. Xu K., Zhang H., Cohen-Or D., Chen B.: Fit and diverse: set evolution for inspiring 3d shape galleries. *ACM Trans. Graph. 31*, 4 (July 2012), 57:1–57:10. doi:10.1145/2185520.2185553. See page 40.

YWR09. Yin X., Wonka P., Razdan A.: Generating 3d building models from architectural drawings: A survey. *Computer Graphics and Applications, IEEE 29*, 1 (1 -2 2009), 20–30. doi:10.1109/MCG.2009.9. See page 35.

ZKT*14. Zmugg R., Krispel U., Thaller W., Havemann S., Pszeida M., Fellner D. W.: A new approach for interactive procedural modelling in cultural heritage. In *Archaeology in the Digital Era: Papers from the 40th Annual Conference of Computer Applications and Quantitative Methods in Archaeology (CAA 2012)* (2014), pp. 190–204. See pages xi, 118, 127, 128, 129, 130, 131, 132, 138, 141, 142, 162, 164, 166, and 167.

ZTH*12. Zmugg R., Thaller W., Hecher M., Schiffer T., Havemann S., Fellner D. W.: Authoring animated interactive 3D museum exhibits using a digital repository. In *VAST* (2012), Arnold D. B., Kaminski J., Niccolucci F., Stork A., (Eds.), Eurographics Association, pp. 73–80. URL: http://dblp.uni-trier.de/db/conf/vast/vast2012.html#ZmuggTHSHF12, doi:10.2312/VAST/VAST12/073-080. See pages xi, 118, 134, 135, 187, 188, 189, 190, 191, 192, 193, and 194.

ZTK*13. Zmugg R., Thaller W., Krispel U., Edelsbrunner J., Havemann S., Fellner D. W.: Deformation-aware split grammars for architectural models. In *2013 International Conference on Cyberworlds* (2013), IEEE Computer Society Conference Publishing Services, pp. 4–11. doi:10.1109/CW.2013.11. See pages xi, 107, 108, 111, 112, 113, 115, 116, 117, 180, 183, 184, and 185.

ZTK*14. Zmugg R., Thaller W., Krispel U., Edelsbrunner J., Havemann S., Fellner D. W.: Procedural architecture using deformation-aware split grammars. *The Visual Computer 30*, 9 (2014), 1009–1019. doi:10.1007/s00371-013-0912-3. See pages xi, 107, 108, 113, 173, 174, 176, 180, and 184.