Johannes Edelsbrunner, Dipl.Ing.

Domain Specific Methods for Procedural Modeling of Historical Architecture

Doctoral Thesis

to achieve the university degree of

Doctor of Philosophy

PhD degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Priv.-Doz. Dipl.-Inform. Dr.-Ing. Sven Havemann

Co-Supervisor

Assoc.-Prof. Priv.-Doz. Ph.D. M.Eng. Alexei Sourin

Institute of Computer Graphics and Knowledge Visualisation
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolf-Dietrich Fellner

Graz, February 2018

Affidavit

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

Date	Signature

Abstract

This thesis is concerned with the question of how to efficiently model and represent historic buildings in the computer. Since detailed 3D modeling can often require considerable amounts of effort, automation is a natural goal to strive for. This automation can be provided by procedural modeling. Common procedural modeling techniques excel at the generation of a vast amount of simple buildings for entire virtual cities. While simple box-shaped buildings can be easily described, for more complex buildings such as complex historic buildings procedural techniques can be used only sparely.

Virtually generated buildings and cities are increasingly demanded by virtual worlds, movies, and video games. Modeling them in detail requires a huge amount of resources and historic buildings are one part that is not well covered today. Historic buildings have different parts that need different modeling techniques.

This thesis investigates some of those parts and tries to find new answers on how to model them procedurally. The introduced modeling techniques comprise a technique to model complex roof landscapes of historic cities, a technique to procedurally model the geometry of round building parts, and a technique to capture the forms of ornamental decorations in historic buildings. Further it compares domain specific methods from software design to procedural modeling techniques and draws comparisons there. The basis for all the research forms a common programming language that is designed for procedural modeling.

Contents

1. Introduction

1.1. Motivation

In computer graphics the generation of 3D models of buildings is an ever present topic. For virtual cities or virtual worlds in general, buildings are an essential part. And usually, the bigger the virtual world, the more buildings have to be generated. This means that more and more artists have to be employed to model these huge worlds.

This is where *Procedural Form Fodeling* (or in short *Procedural Modeling*) comes into play. Procedural Modeling enables the generation of 3D models via computer programs. Past works in this field have already achieved to generate big cities with an astonishing amount of buildings. The major advantage of Procedural Modeling is that repeating forms/arrangements/patterns can be represented in a computer program, and then be instantiated manifold in a scene. Many box-based houses and skyscrapers are already reproducible with a lot of detail (see Figure 1.1). However, the methods have limitations when it comes to the generation of more complex buildings.

One category that is hard to model are complex historical buildings (see Figure 1.2). There is a wide variety of historical architectural styles, and almost all of them have rules that guide the arrangement of architectural elements. These rules are often concerned with repetition and symmetry, which makes these buildings ideal candidates for Procedural Modeling to reproduce. Their more complex shapes, compared to standard box-based houses and skyscrapers, are more difficult to cover with existing techniques.

This motivation leads to the fundamental research question in this thesis:

What can be novel techniques that allow complex historical architecture to be procedurally modeled in the computer?

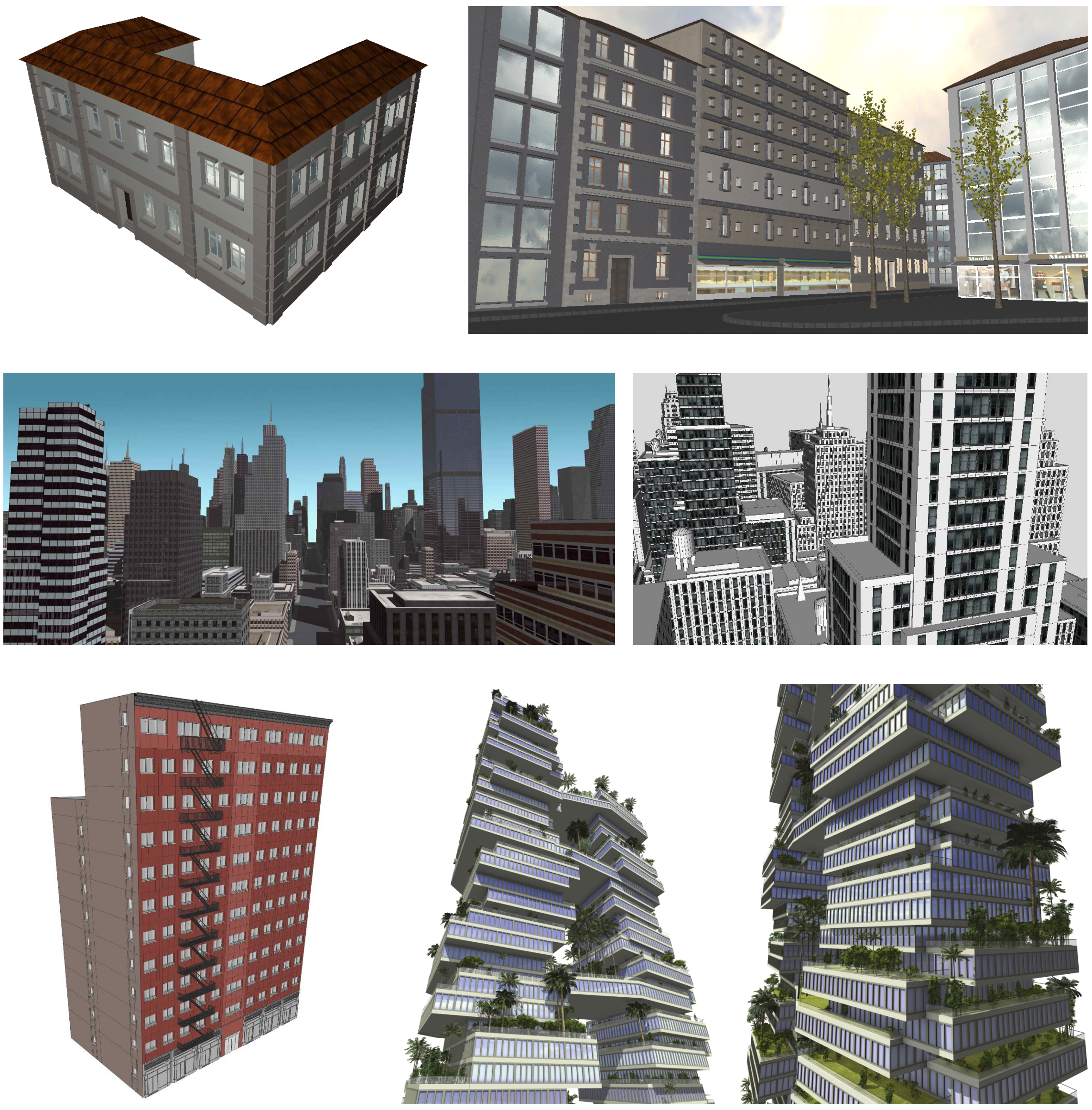


Figure 1.1.: Architecture generated with existing split grammar approaches. Images taken from [96, 65, 77].

(a) St. Peter's Basilica

(b) Karlskirche

(c) Neuschwanstein Castle

(d) Leaning tower of Pisa

(e) Pantheon

(f) Gate of Honour in Versailles

(g) Palace of Versailles

(h) Aerial view of Venice

Figure 1.2.: Complex historic architecture. Many parts of those buildings cannot be modeled with existing procedural modeling techniques. Images taken from [70, 54, 95, 17, 91, 22, 89] and ©2018 Google, Map data ©2018 Google.

1.2. Why Procedural Modeling?

Since nowadays the geometry and color information of buildings can be scanned with many different methods, the question arises why buildings and entire cities are not simply scanned to be brought to life in the computer? In fact, this is already done by mapping applications like google maps. A scan of a building or a city usually results in a triangle mesh with corresponding textures. But this triangle mesh does not have any semantics. This means it has no information where a door, a window, or even a building is in the whole model.

Semantics in 3D Models

The study of semantics is in general concerned with the meaning of things. In the context of 3D modeling this could be the meaning of different parts of a 3D model. Without semantics a 3D model is only a form - a mesh, a collection of primitives, or a set of equations describing form - but different parts of a model cannot be identified, queried, or interacted with in a meaningful way. Elements of a 3D building model such as doors, walls, and windows might be visually recognizable by humans, but not by the computer when there is no semantic information present. This limits the set of use-cases for models without semantic information. For example, in a semantic 3D model doors can be replaced by other doors, and windows can be replaced by other windows - or possibly walls.

The semantic model can also capture the construction information for a building. In order to plan, organize, and construct a building this information is critical. Based on this information plans and extrusion plans for a building can be generated, semantic building models can be queried for parts such as number and location of windows, or simulations and optimizations for factors like heat distribution can be run.

In contrast to a scan of a building, a procedural model can now provide and generate this semantic information through the execution of its program or rule set.

Procedural Modeling as a Method to create 3D Models

Procedural Modeling in general describes a wide field of techniques that create 3D models from some form of program or rule set. The creation of a Procedural Model is not without cost. Usually there is much more effort involved in creating the procedural model compared to simply modeling the object with a standard 3D modeling program. The additional effort pays off when the model needs to be modified later. Since procedural models are generated from a program or rule set, those can be parameterized, and by providing different parameters, different manifestations of the same model can be generated.

This brings a lot of flexibility to a model that is generated procedurally. Most traditionally modeled models consist only of the geometry in form of a triangle mesh and are often impossible to modify sufficiently when requirements change. But besides flexibility this also allows to instantiate different versions of the same model (with different parameters) multiple times. This enables the generation of vast virtual worlds since buildings in one city (or world) usually share many commonalities between them, while each building might still be unique and therefore cannot be a simple copy of some pre-modeled triangle mesh.

Combining Procedural Modeling with other Methods

So far, the reader might get the impression that the choice of Procedural Modeling versus any other modeling method is an exclusive one. However, this is not the case. Procedural Modeling can be combined with many other methods. For example, a triangle mesh can be created by a 3D scan of a building, and special algorithms can be used to infer parameters for an existing procedural model that describes general buildings. With these parameters the procedural model generates a 3D model that resembles the scan up to a certain accuracy. With the added benefit of being able to generate all the valuable semantic information, and having a modifiable model that can be changed according to given requirements.

1.3. Outline

The aim of this thesis is now to investigate various existing procedural modeling systems, their design, what can be modeled with them, and what is still missing for procedural modeling of different domains of historical buildings.

Based on the missing capabilities then, new domain specific methods are developed that try to fill the gaps that previous techniques left.

Chapter 2 will give an overview of existing research for procedural modeling. Various papers will be examined and methods with which they produce models will be highlighted. Then shortcomings in the context of modeling historical architecture will be illustrated.

Chapter 3 then identifies key areas that could be improved and develops a plan to solve some of the shortcomings from the previous chapter.

Chapter 4 will be a short prerequisite, giving an introduction into domain specific languages and their usage in software design. This will later serve to analyze the different techniques.

Chapters 5, 6, and 7 then present case studies of novel work that has been done. The case studies show how domain specific procedural approaches can be used to model parts of historic buildings.

Chapter 8 will give an evaluation of the newly presented modeling methods and show what parts of historic buildings can be procedurally modeled with them. It will also analyze further properties of the methods and give insights into the creation of procedural methods for modeling domains.

Chapter 9 will wrap up and show contribution, benefit, and validation of the presented new methods.

The closing Chapter 10 will finish with an outlook on possible future work.

2. Procedural Modeling of Architecture

2.1. Introduction to Procedural Modeling

Procedural modeling promises to ease the creation of 3D models by automating the modeling process.

Many 3D models are traditionally made by artists using advanced graphical 3D software by manually modeling the geometry. This allows the artist a high level of control to instantly see the result of the modeling operations that are performed.

Procedural modeling on the other hand is often less straight-forward. It is a wide field that uses a variety of techniques to describe and generate 3D models. For example, scripting languages, or special visual procedural modeling and editing tools. These techniques can require special knowledge, time and effort to learn them, and are not always very easy to use or do not give immediate and easy to interpret feedback. Also, the needed developing effort for specialized procedural modeling tools can be substantial.

But procedural modeling brings one fundamental concept of computer science into 3D modeling which is not yet covered by traditional modeling tools in its full extent: automation.

The fine grained control for the creation of 3D models of graphical 3D modeling software is unparalleled by procedural modeling systems. But what if variations of a model have to be created dozens or thousands of times? Procedural modeling allows the generation of countless buildings for whole cities for movies, games, or urban planning. Even whole universes of generated planets including their environments for video games (for example in the case of the recently released video game 'No Man's Sky').

Whereas traditional modeling is mostly concerned with the final shape of an object that the artist creates, procedural modeling might also be concerned with, or even need, the information about the inner structure of the object, the dependencies of its various parts, or the blueprint that describes the way the object is built. This information allows the definition of parameters that can influence parts or dimensions of the object, and with these parameters the procedural system is able to generate variations of the object.

How the procedural modeling system is designed is heavily dependent on the kind of objects that should be generated (the domain). Different systems have different advantages and disadvantages and are suited better or worse for a particular domain.

2.2. Survey of important existing Works for Architecture

This section will give an overview of some of the exiting and most important works in the field of procedural modeling of architecture.

2.2.1. Procedural Modeling of Cities

In their paper *Procedural Modeling of Cities* [69], Parish and Müller present a system for modeling cities where extended urban areas may be generated from minimal input data. The problems associated with city modeling in computer graphics arise from the huge amount of geometry required, making it unfeasible to consider modeling them manually. However, it is also true that urban environments generally develop according to systems of clearly observable rules, such as those governing the placement of buildings and transport networks (roads, highways and the like). This means that cities are ideally suited as subjects for procedural modeling. Parish and Müller observe that placement of streets and highways tend to arise out of the population density and local environmental factors, as well as deliberate planning to a set pattern. Placement of buildings occurs in accordance with considerations as to the aesthetics of the building, the historical style of the area and the statutory requirements set by the city for buildings in that particular area.

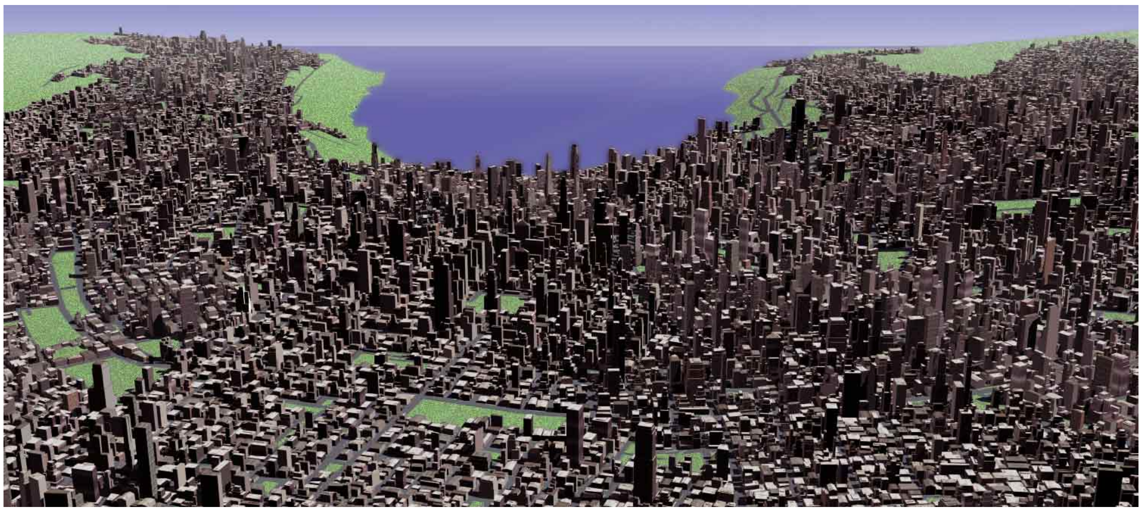


Figure 2.1.: Paper: Procedural Modeling of Cities [69]

Their system is mostly based on L-Systems, an older technique already used in [72, 60, 73]. First, they use different input image maps (like land and water boundaries, population density). Then they generate a system of highways that adapts to the requirements of to these maps. Streets are then generated via an L-System in the regions bounded by the highways. The streets again partition the space into blocks, where then the buildings are generated via a different L-System.

The different stages in detail:

Input Data As input data they do not need aerial images of streets or buildings, but only use different image maps. They have maps for geographical data such as elevation and geographical type (e.g. land, water, vegetation, etc.) and sociostatistical data such as population density, zone type (e.g. residential area, commercial area, etc.), street patterns (e.g. rectangular, radial, etc.), and maximum building height. From this input data, an infinite amount of large-scale cities can be produced.

Streets For the streets they distinguish two types: regular streets and highways. Highways connect areas with high population density globally, and streets then span the areas between highways locally. The highways are generated by an

L-System which scans a density input map for high density areas, and orients the highways to these areas. After the highways are generated, an L-System generates streets that cover the areas between highways according to local population density, and ensure that everywhere exists access to highways. If a street leads into a water area or a park, local modifications can be made. They use self-sensitive L-Systems that allow the branches of the L-System to grow together, which is essential for the street network. This changes the created topology of the L-System from tree-like to net-like.

For the pattern of the streets they provide four different rules that can be chosen:

- *Basic*: The streets simply follow population density.
- *New York*: The streets follow a rectangular grid aligned to a specific direction.
- *Paris*: The streets follow radial tracks around a center.
- *San Francisco*: The streets follow the shallowest elevation, connect by shorter streets that follow the steepest elevation.

Building lots The street network partitions an area into a multitude of blocks. These blocks are then further subdivided into building lots, until their area is smaller than a user-defined threshold. Only convex lots are generated, and lots which are too small or do not touch a street are removed.

Building geometry In every lot, one building is created. The building geometry is created via a different L-System that takes an arbitrary ground plan and then performs these operations on it: transformation, extrusion, branching, and termination. The system also uses pre-build geometry for objects such as roofs, antennas, etc.

Building textures The facades of the buildings are created with semi-procedural textures. Facades often exhibit one or multiple grid-like structures, where most of the cells have the same objects in them (for example windows or doors). They chose an approach where they have multiple facade layers which are combined to one facade. One layer is comprised of two one-dimensional interval groups. The intervals are used for the alignment of objects such as windows and doors. They can also define specialized information for rows, columns, or cells in a layer (for example, to make ground floor windows bigger).

2.2.2. Instant Architecture

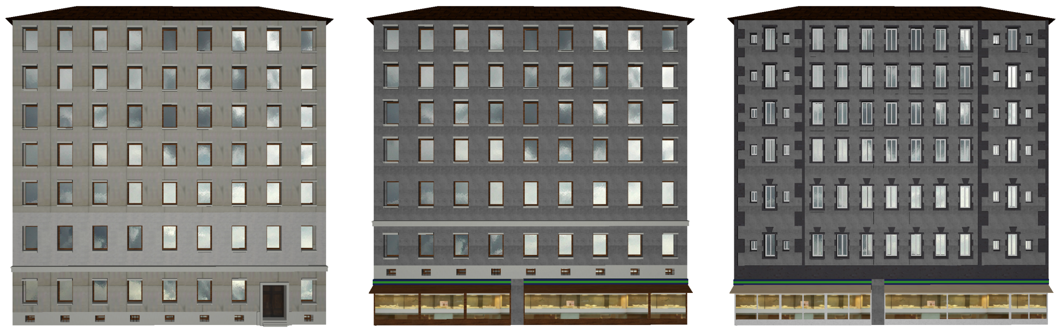


Figure 2.2.: Paper: Instant Architecture [96]

In contrast to *Procedural Modeling of Cities* (Section 2.2.1), which presents a system for the generation of entire cities, Wonka et al. [96] focus fully on the modeling of buildings in their paper *Instant Architecture*. They generate building facades via a specialized grammar approach, which is a further development of the shape grammars of Stiny [79]. They use a split grammar that generates the 3D geometry of the facade, and a control grammar that sets attributes like material or color for the geometry.

With the grammars an extensive database of rules is built. When a facade is generated, an automatic process can choose suitable rules in each derivation step of the grammar. This way there is no need for a specialized grammar for each building. A variety of designs can be created out of the rules in the database. Which rules are chosen can be random or influenced by user defined design goals.

The basis for the geometry creation are simple geometric shapes that are then divided into smaller shapes by the split grammar. Properties such as the material of elements are transferred to the newly created shapes. How this is done is determined by the control grammar. For the new shapes, only rules that ensure either a plausible result, enough variation, or the achievement of user-defined design goals, are chosen. Additionally, for special elements, also pre-modeled assets are used.

The presented approach is based on two-dimensional placement of elements in a grid-like fashion, and can model many facades that follow a mostly regular layout. The approach is limited to modeling of facades though, and for modeling of building shells other techniques have to be used.

2.2.3. Procedural Modeling of Buildings

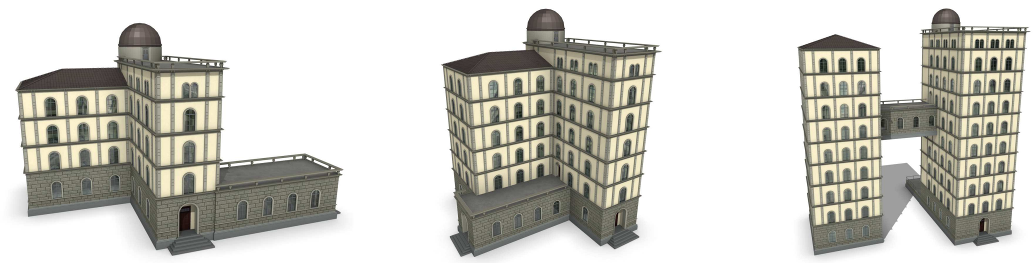


Figure 2.3.: Paper: Procedural Modeling of Buildings [65]

An innovative split grammar called CGA shape presented in the paper *Procedural Modeling of Buildings* [65] by Müller et al., allows users to surmount some of the challenges associated with producing large quantities of high quality and intricately detailed buildings through procedural modeling and the use of context sensitive shape rules. Their work is based on *Procedural Modeling of Cities* (Section 2.2.1) and *Instant Architecture* (Section 2.2.2), and they point out some limitations of these previous works. In their opinion, *Procedural Modeling of Cities* cannot create enough detail and produces unwanted intersections of architectural elements, and *Instant Architecture* has split rules that are suited for simple mass models, but complex models require an excessive amount of them. They design their split grammar such that buildings are not restricted to axis aligned shapes, roof surfaces and rotated shapes can be created by the grammar, and they give a clear definition of how shape rules are defined.

Their notion of a context sensitive split grammar primarily means two techniques:

- *Occlusion tests* - The split grammar can detect when elements are occluded, for example when a part of a building intersects with the window of another part. Then the window can be modified or skipped completely.
- *Snapping* - The position and size of shapes can adjust to so-called snap lines. This way, for example, windows of one part of a building can align with windows of another part.

A shape in the split grammar has a symbol, attributes, geometry, and a scope. The scope defines special geometric information: the position of the shape, a coordinate system in which the shape lives, and the size of the shape.

The form of the production rules is explicitly defined in the paper. The rules are in human readable form and are intended to be written or reused by users that not necessarily have to be programmers. A rule has the following form:

id: predecessor : cond $\rightarrow$ successor : prob

with:

- *id*: unique identifier for rule
- *predecessor*: shape that is being replaced
- *cond*: necessary condition that has to be fulfilled
- *successor*: new shape (or multiple shapes)
- *prob*: probability for choosing this rule

There are different ways to specify the successor, but maybe the most important ones are the split operators. These split the actual shape along one or multiple axes:

- *Subdiv*: Subdivides a shape into multiple ones. The width of subdivisions can be specified and is either absolute or relative.
- *Repeat*: Repeats a new shape of certain width as often as possible (according to the given space) in the current shape.
- *Component*: Replaces a shape with its individual components (faces and edges).

The split operators are mainly responsible for creating the geometry of the facades. Additionally pre-modeled assets are used, for example for cornices.

The procedure for modeling a building in the split grammar is usually as follows: First, the rough outer hull is created by a combination of simple shapes (for example boxes, cylinders, etc.). These simple shapes already come with predefined roof geometry. There can be different types of roofs such as hipped, gabled, mansard, or cross-gabled roofs. Then the faces of the building are extracted to form the basis for the facades and roof surfaces. There the split rules are applied. Occlusion tests and snap lines then modify the geometry as necessary.

Different types of buildings such as single family homes, office buildings, and skyscraper can be generated with the method. In the paper, different examples of buildings and cities are shown. One is the reconstruction of the ancient Pompeii, one is an urban city modeled by a professional modeler (but not programmer), and one is a residential area inspired by Beverly Hills.

While *Instant Architecture* (from Section 2.2.2) is concerned with two-dimensional placement of elements of building facades, this approach also deals with the modeling of building shells by combination of simple shapes with roof geometry. However, the design of the facade elements is again done in grid-like fashion, and the roofs are static pre-modeled meshes, which limits the application of the approach for complex forms.

2.2.4. Advanced Procedural Modeling of Architecture

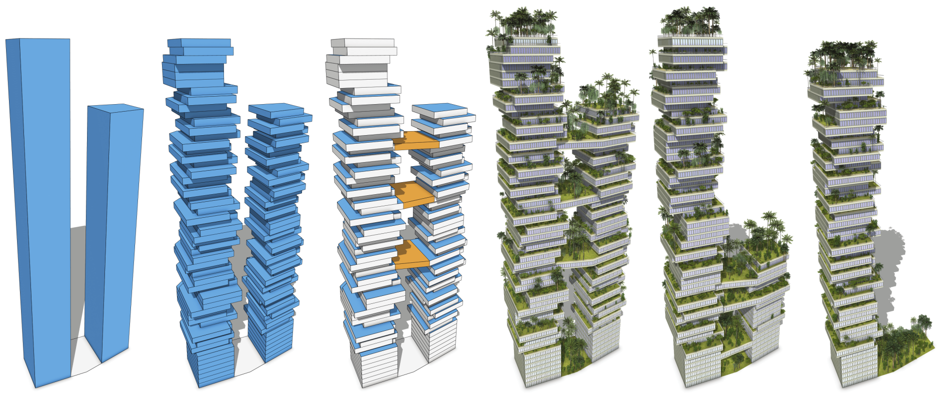


Figure 2.4.: Paper: Advanced Procedural Modeling of Architecture [77]

Schwartz and Müller discuss the limitations of the split grammar language CGA (described in 2.2.3) and present their extension to this language in the paper *Advanced Procedural Modeling of Architecture* [77], which they call CGA++. According to their argument, this development is necessary in order to be able to approach context sensitive tasks, something which is currently impossible using CGA or other existing systems. A lack of available information about other shapes in proximity to the shape being refined is a drawback in these systems and limits their expressiveness. In comparison, shapes and shape trees in CGA++ are assigned first-class citizenship, which enables such functions as:

- Direct access to shapes and shape trees
- The option to perform operations on groups of shapes

- The ability to rewrite the structure and hierarchy of shape trees
- Capacity for automatic generation of new shape trees

Another innovative aspect is the event-driven system that allows coordination across multiple shapes using a dynamic grouping and synchronization mechanism.

Current systems such as CGA shape have several limitations in their expressiveness:

- *Multiple shape refinement coordination*: Any decision that is made in the shape tree can only affect child shapes of the current shape. If the decisions should be influenced by properties of the child shapes, these properties have to be inferred manually beforehand.
- *Operations on multiple shapes*: Operations taking multiple shapes, like boolean operations, are not available.
- *Contextual information*: No contextual information, for example for alignment is available.
- *Spawning derivations in derivation*: There is no mechanism to create a separate derivation tree in the current derivation. This can be helpful for example for choosing between multiple alternatives of derivations.

The paper overcomes this limitations by two new main language features: *Shapes as first-class citizens* and *Events*.

Shapes as first-class citizens This means that individual shapes can be uniquely identified, passed around, stored as values, and used as parameters for operations. The shape tree supports different operations for accessing, traversing, and querying it. Additionally, temporary versions of the shape tree can be generated.

Events Events serve as a grouping and synchronization mechanism. They provide coordination across groups of shapes by enabling the exchange of information, and enabling a consistent decision of how to proceed in the individual shapes. They also make it possible to influence the derivation order, which can ensure that certain shapes are available for shape queries.

In addition to this two new main features, CGA++ extends the grammar language itself with various constructs from general programming languages, in order to make it more versatile. Among other things, different programming constructs such as booleans, numbers, strings, lists, tuples, and functions are available.

The authors give different scenarios of applications where CGA++ gives additional expressiveness over other methods:

- *Urban planning*: Division of areas into building lots and creation of buildings on them.
- *Buildings*: Creation of buildings with interconnected structures.
- *Facades*: Alignment of elements (windows, doors) according to other elements.

2.2.5. Shape Grammars on Convex Polyhedra

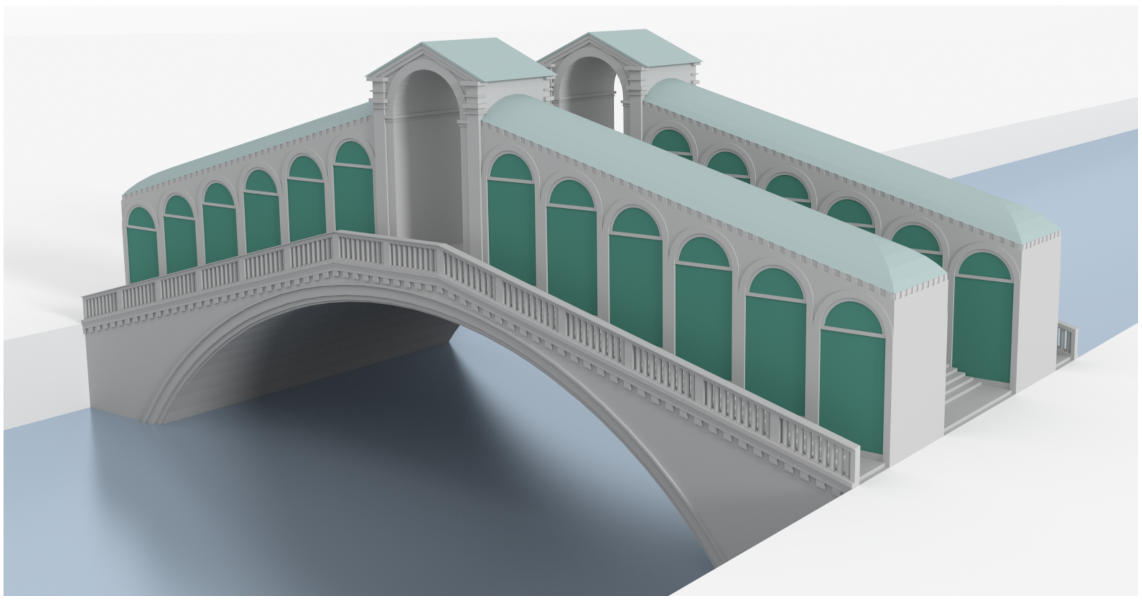


Figure 2.5.: Paper: Shape Grammars on Convex Polyhedra [87]

As mentioned, many shape grammar systems use the notion of a scope which defines the size of the current shape as an axis aligned box. Using a box as a bounding volume for a shape is practical because it can be split along any of the main-axes into smaller boxes. However, as Thaller et al. in *Shape Grammars on Convex Polyhedra* [87] argue, this limits the expressiveness of the forms that can be generated with the shape grammar. They generalize the bounding volume from a box to a convex polyhedron. There the split operations can be performed in arbitrary directions and not only along the main-axis. The rules of the split grammar can automatically adapt the split operations to the space that is given by the convex polyhedra. The paper explains how the common split operations are generalized for convex polyhedra and introduces new shape operations.

Because the system generalizes the boxes in the scopes to convex polyhedra it can be seen as a new Non-Terminal class in the sense of the later Section 2.2.7. The geometry itself is stored separately though, and can also be non-convex. The split operations orient themselves to the convex polyhedra and split both, the convex polyhedra and the geometry.

Since the convex polyhedra are volumetric shapes, they can be further refined by splits and can adapt to the space given by their surrounding. This enables more expressiveness than simple meshes that are used as non-terminal replacements, such as in previous methods.

The most prominent split methods of this system are:

- *Plane split*: Divides a shape by one arbitrary plane
- *Subdivision and repeat splits*: Work as in previous methods, but can now operate along arbitrary directions, or even curves.
- *Frame split*: Divides a shape into a smaller inner shape, and multiple outer shapes along the border (useful for example for windows and their frames).

A concept that the paper discusses is that of *procedural assets*. Instead of placing pre-modeled assets into the boxe-shaped scopes of classical Non-Terminal shapes, the geometry itself can now be modeled via sculpting of the convex polyhedra with split operations. Convex polyhedra can often approximate the wanted geometry in sufficient detail. This allows further refinement of the resulting geometry as needed. Classical architecture can be one area where this approach can be used to model the detailed geometry. Further split operations for this concept are:

- *Radial split*: Partitions a shape into 'cake slices' (for example for round windows).
- *Polyline split*: Splits a shape along a series of planes that are aligned on a polyline.
- *Arch*: Divides a shape into two shapes for an arch. One for the brickwork, and one for the hole inside (for example for arches in historical architecture).

The paper *Shape Grammars on Convex Polyhedra* [87] was written by the work group for Procedural Modeling at the *Institute of Computer Graphics and Knowledge Visualisation (CGV)* [46] at *Graz University of Technology*.

2.2.6. Procedural Architecture using Deformation-Aware Split Grammars

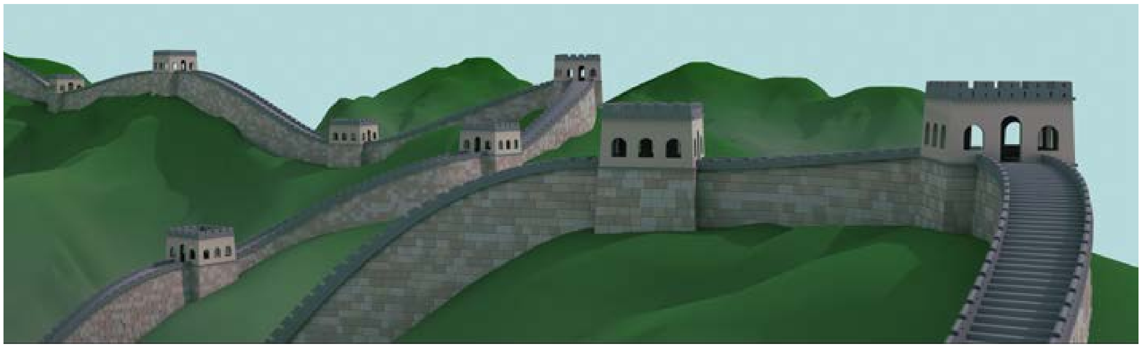


Figure 2.6.: Paper: Procedural Architecture using Deformation-Aware Split Grammars [98]

One of the limitations of previous split grammar systems is the fact that the only method of approximating the curved features of buildings was via static pre-modeled assets, with rules applying solely to planar surfaces. In contrast, the new system presented by Zmugg et al. in the papers *Deformation-aware split grammars for architectural models* [97] and *Procedural architecture using deformation-aware split grammars* [98] allows free form deformations to be defined in the grammars. The deformations are not merely a post-processing step that deforms the whole model, but can be defined at any level in the split grammar and thus the split operators can adapt the splits to the deformed space. The authors show different types of splits where the split planes are either deformed by the deformations, or remain straight but still split the deformed geometry.

Traditionally, split grammars use planar surfaces for the splits. However, also curved structures can be suited for a grammatical representation. For example, a stonewall has a regular and repeating substructure (the bricks in the wall) and can follow an arbitrary path. If the path is curved, this results in an overall curved structure. The typical approach is to model it (for example with shape grammars) and then deform the geometry afterwards. However, this has limitations since the grammar rules cannot adapt to the deformed space, and deformed and undeformed geometry cannot be mixed. Therefore, in this system, the deformations can be defined in any rule of the grammar. This deforms the scope and the geometry of the current shape and its sub-shapes. The Non-Terminals contain geometry, a bounding shape, and a list of arbitrary free-form deformations. The free-form deformations can be nested so they are stored in a list of multiple nested deformations. The actual process of deforming the geometry is done after the grammar is evaluated with the help of this list.

There are three different types of deformations described in the paper:

Deformed Splits on Deformed Geometry Here the splits are defined by planes on the undeformed local coordinate space. It is implemented by performing classical straight splits on the undeformed geometry and then deforming the result afterwards (leading to deformed split surfaces). The resulting sub-shapes of a split are also annotated with the same deformation.

The measurements for the splits should be constant, independently of the applied deformation. Therefore, they are taken in the already deformed space, but are transformed back into undeformed space in order to allow the straight splits on undeformed geometry. The common split operations are deformation aware, meaning that the subdivide split adapts the sizes of the sub-shapes automatically to the deformation, and the repeat split adapts the number of sub-shapes automatically to the deformation.

Straight Splits on Deformed Geometry Here the splits are performed with actual straight planes on the already deformed shapes. An example would be straight window panes in a wall that is deformed. Therefore the *bake* operations is introduced, which instantly deforms and solidifies the geometry before it is split (with straight planes). The bounding shapes of the scopes now become an issues. A classically used box gives a very poor approximation of the deformed shape, therefore the authors suggest to use at least convex polyhedra as described in Section 2.2.5, which can also be split in any arbitrary direction.

Deformations of Adjacent Objects When two parts are deformed differently and they are adjacent to each other, special handling of the seams where they meet is often required. At the seams where the parts meet, they can now be detached or intersect each other after the deformation. Therefore, either additional geometry has to be created, or existing geometry has to be trimmed with boolean operations.

The authors also describe a method for the creation of houses in cartoon style where the walls are all deformed independently. The method works by automatically creating walls from an input ground polygon and saving the adjacency information for them. Then an automatic procedure corrects the geometry at the seams of the walls.

The papers *Deformation-aware split grammars for architectural models* [97] and *Procedural architecture using deformation-aware split grammars* [98] were written by the work group for Procedural Modeling at the *Institute of Computer Graphics and Knowledge Visualisation (CGV)* [46] at *Graz University of Technology*.

2.2.7. Generalized Use of Non-Terminal Symbols for Procedural Modeling

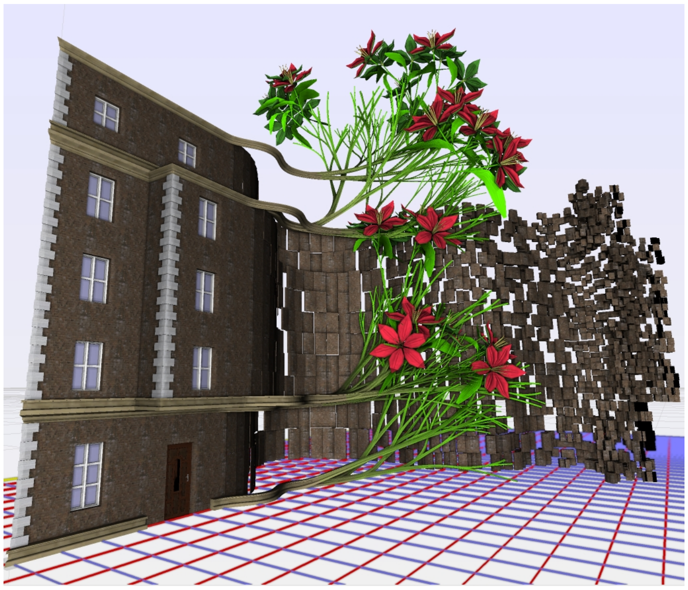


Figure 2.7.: Paper: Generalized Use of Non-Terminal Symbols for Procedural Modeling [51]

Krecklau et al. present an adaption of the common split grammar languages (for example as described in Sections 2.2.1 and 2.2.3) in *Generalized Use of Non-*

Terminal Symbols for Procedural Modeling [51]. Their language G^2 aims to extend the expressiveness of previous approaches while providing a simple and easy to read syntax. The resulting grammar allows for the inclusion of Non-Terminal objects and symbols of various types. These may be assigned attributes and operators that are specific to the domain, and may also act as parameters in the definition of modeling rules. The ability to create templates for abstract structures is convenient for the user who may wish to re-use them elsewhere in the grammar. These extensions assist in the modeling of complex objects such as architecture and plants, as is exemplified in their illustrations. The language itself is derived from Python.

The new main features of the system are:

- *Abstract Structure Templates*: These are rules that take Non-Terminal symbols as input parameters. This allows the reuse of rules in different contexts (similar to higher-order functions in functional programming languages).
- *Non-Terminal Classes*: The Non-Terminals are not limited to boxes in this system, but can be user specified. Each Non-Terminal class has its own operators and attributes. The aim is to bring procedural modeling closer to conventional modeling, since conventional methods can be emulated with the Non-Terminal classes. An example of a free form deformation as a Non-Terminal class is given in the paper, which allows modeling of deformed objects.
- *Flags*: They are used to identify individual parts of the grammar derivation tree, where then specific rules can be applied. Through this, no dependencies between rules are introduced.

2.2.8. Component-Based Modeling of Complete Buildings

Leblanc et al. present a component based approach to the problem of procedurally modeling architecture in *Component-Based Modeling of Complete Buildings* [53]. Architecture has many interdependent elements and the method described in this paper attempts to create a system of representation through the use of components, which are elements that have been defined in terms of space and semantics. The system works with a tree of components that the user can query. The components from the query results can then be modified via shape operators, and new components can be generated. This results in an process that iteratively refines the component tree.

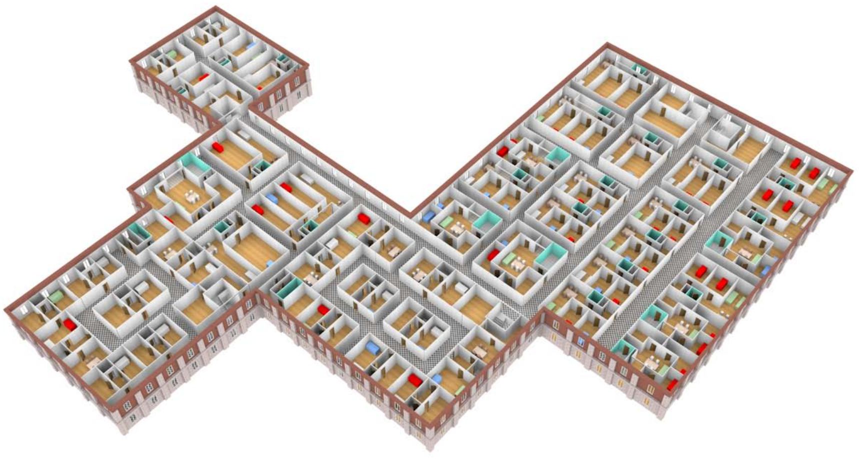


Figure 2.8.: Paper: Component-Based Modeling of Complete Buildings [53]

The system is able to generate both the exterior and interior of an entire building. This is a special property of the system, since previous systems were often good in one method, especially the exterior, but few showed capabilities to combine both. It is using methods like split grammars and constructive solid geometry in order to generate the final model. These methods are applied via special operations that work on a set of shapes. A query mechanism lets the user query all of the shapes and get back a set of shapes to work on. Operations and queries for a model are specified via a programming language which is intended for designers with programming skills.

2.2.9. Creating Procedural Window Building Blocks using the Generative Fact Labeling Method

Thaller et al. present the *General Fact Labeling Method* [85], a method that aims to organize the process of shape analysis and shape synthesis. Beginning with a finite number of 3D shapes (independent of whether real or virtual), they attempt to identify a small group of simple and combinable functions that can be combined to represent the given shapes.

A window can be seen as a combination of inter-related design elements. Win-

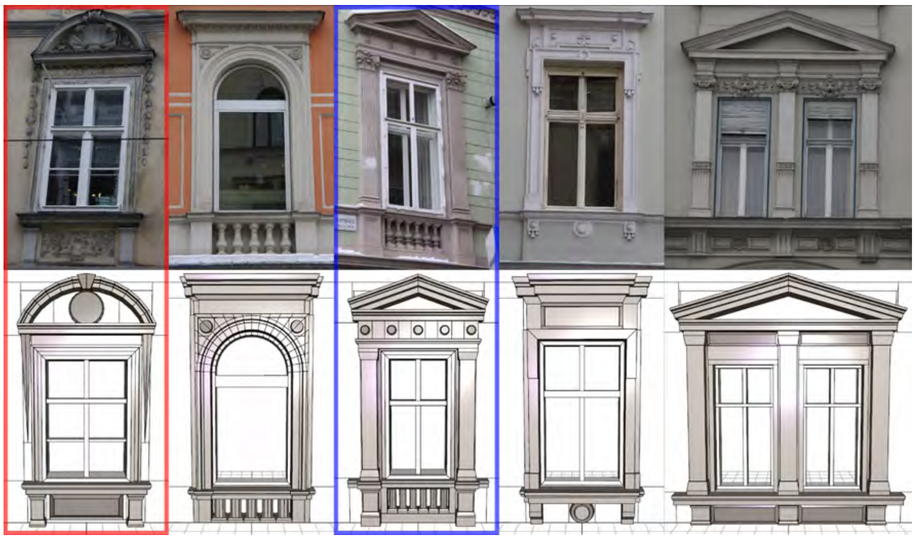


Figure 2.9.: Paper: Creating Procedural Window Building Blocks using the Generative Fact Labeling Method [85]

dows can pose a complex problem for 3D modeling because there can be vertical or horizontal coherence between windows next to each other.

For 3D reconstruction of urban environments there is often one of the following problems present with windows: Either the window models have good quality, but are not matching the original window (as in the case with pre-modeled assets for windows), or the window models match the original windows, but their quality is not very good (as in the case with simple textures for each window).

The General Fact Labeling Method aims to produce windows that are both, high quality and closely matching the original. It consists of three phases:

1. *analysis*: structure elements into fact labels
2. *synthesis*: library of composable procedural assets (corresponding to elements)
3. *verification*: 3D reconstruction of windows and comparison with originals

The goal of the method is not only to recreate the original exemplars, but also to create new exemplars in the design space.

Fact Labeling Process The fact labeling process is proceeding from coarse to fine. First, rough structural units, like window layouts are labeled, then local parts and refinements are labeled.

The classification scheme aims to group observations so that they are procedurally composable. Different elements that could be used for classifications can be for example: window count, window side elements (columns/pilasters), window sill, area above window (cornice/pediment), frieze (frieze/architrave), window layout (interaction between pillars/frieze/architrave), window shape (rectangular, round, etc.), window frame, pediment, or cornice.

The paper *General Fact Labeling Method* [85] was written by the work group for Procedural Modeling at the *Institute of Computer Graphics and Knowledge Visualisation (CGV)* [46] at *Graz University of Technology*.

2.3. Restrictions and Room for further Research

The presented methods so far are sufficient to generate big amounts of buildings with mostly box-based or prism-based geometry for a whole city (as shown in 1.1). But when it comes to more sophisticated forms they reach their limits. Some methods (most notably those presented in Sections: 2.2.5, 2.2.6, and 2.2.7) extend the possible classes of forms that can be generated, but they are still not able to fully generate all kinds of architecture. Especially complex historic architecture has many forms that cannot be adequately modeled by existing methods. Still, it often has a lot of repetition and forms that adhere to strict rules, which should make it a suitable target for procedural modeling.

The following is a list of examples of well known historic architecture (images can be found in the introduction in Figure 1.2) where existing approaches have specific problems:

- *St. Peter's Basilica:* It is one of the best known examples of Renaissance and Baroque architecture. It is a very complex structured building and therefore difficult to model efficiently using split grammars. The domes of the building, due to their spherical arrangement of geometrical structures, cannot be modeled efficiently with common split grammars.
- *Karlskirche:* Similar to St. Peter's Basilica and an example of the Baroque and Rococo styles. Also with a large dome and many round elements.

- *Neuschwanstein Castle:* The castle, built as Romanesque Revival building, features many round towers that cannot be adequately reproduced by existing techniques.
- *Leaning tower of Pisa:* The tower in the Romanesque style has many repeating structural elements on its outside, and is therefore a suitable candidate for procedural modeling. The circular arrangement of the elements makes this very hard for common split grammars though.
- *Pantheon:* The huge dome of the ancient roman Pantheon is coffered in a spherical arrangement in the inside. Additionally, the support structures and walls beneath the dome are arranged circularly. Again, almost impossible to model with common split grammars.
- *Gate of Honour in Versailles:* The entrance to the Palace of Versailles is formed by a gate with ornamental forms. Ornamental forms are generally hard to cover with modeling techniques today. The challenge are free-form shapes and shapes following a curve, which are present in many ornamental wrought irons.
- *Palace of Versailles:* The Palace (in the French Baroque style) itself consists of multiple wings and parts that are connected. This results in a complex roof structure which is hard to model with existing procedural modeling techniques. It also has a lot of ornamental decoration which so far can only be provided via pre-modeled assets to procedural modeling techniques.
- *Venice roof landscape:* Venice, as many other old historic cities, consists largely of a collection of small and simple houses that are more or less arranged irregularly. The roofs of these houses form a very complex roof landscape, that is hard to reproduce accurately.

3. Research Hypothesis

The examples given in Section 2.3 show, that while many standard buildings and skyscrapers of cities can be modeled by procedural modeling techniques, buildings of historic cities common in Europe, but also all over the world, cannot sufficiently be modeled.

The first hypothesis for modeling historic buildings can now be postulated:

- **H1**: *Historic buildings cannot be reasonably modeled with existing procedural shape modeling techniques.*

In order to model historic buildings, different problem domains have to be tackled. This leads to the idea of having different domain specific procedural modeling methods for the different shortcomings that existing procedural techniques have.

3.1. Deconstructing Architectural Models into Domain Problems

Domain specific methods comprise a wide variety of techniques in software development. A domain specific method should be tailored explicitly to model a specific software domain, and it should only be as complex as needed to solve problems in the domain. Experts of the domain, that need not be professional programmers, should be able to use it, or at least be able to participate in the software development process.

The design of procedural modeling techniques is on the one hand concerned with the generation of the model geometry, but on the other hand with the structure of the rules or commands that are used in the procedural description or program. This has a strong link to the field of software design, which deals with the structure of computer programs. Domain specific methods and languages are one strategy to organize a computer program and divide it into ideally independent domain specific components in order to manage the existing complexity and make it accessible to non-programmers. Therefore, bringing domain specific approaches to procedural modeling can help in managing the sometimes vast amount of rules or commands that have to be specified in order to create a procedural model.

DSLs are often much easier to use than normal programming languages. In fact, there is often no need for the user of the DSL to be able to program in a traditional sense. This ease of use gives DSLs the great advantage that domain experts which are not programmers can use them themselves.

At the institute for Computer Graphics and Knowledge Visualization we had an interesting observation in one of our projects where the task was to reconstruct a multitude of windows from the "Gründerzeit". There, domain experts who are educated art historians preferred to use the DSL describing the windows instead of a graphical user interface where they could compose the windows. With the DSL they could copy DSL code from one window and modify it for the next one. In contrast the graphical user interface required them to model each window from scratch.

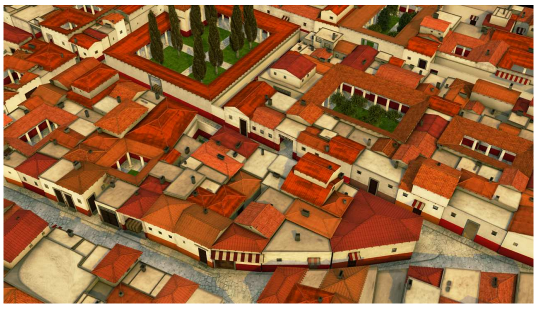
This observation motivated the next research hypothesis:

- **H2**: *DSLs are well suited for the extension of procedural shape modeling techniques to model historic buildings.*

3.2. Unsolved Problem Domains

The following sections present an overview of particular problem domains in procedural modeling of architecture that hinder the creation of procedural models of buildings such as those shown in Section 2.3, and give sketches of ideas of how these problems can be solved.

Through close inspection of a number of historical buildings we have identified in three areas where improvements are particularly beneficial, as they occur in almost every historical building considered (see Section 2.3). Furthermore, they require three different sorts of extensions to the existing formalism, so they have great exemplaric value. This can be formulated as an hypothesis as follows:


(a) Procedurally generated roof landscape. Image taken from [65].

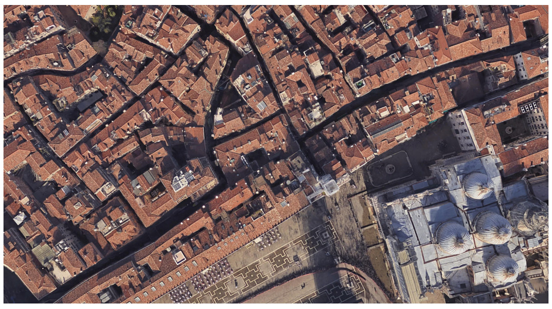

(b) Aerial image of Venice, Italy. Imagery ©2018 Google, Map data ©2018 Google.

Figure 3.1.: Procedurally generated roof landscape (a) compared to an aerial image of Venice (b). A very fine grained and complex roof landscape presents itself in cities like Venice. Existing Procedural Modeling techniques have a hard time producing correct details for such roof geometries.

- **H3**: *The three most effective extensions for historical buildings are:*
- **H3.1**: *Roof landscapes*
- **H3.2**: *Round building geometry*
- **H3.3**: *Free form curves*

Extensive work on new ideas for these problems is then presented in the later Case Studies in Chapters 5, 6, and 7.

3.2.1. Roofs

Roofs in existing procedural modeling systems are often created with the straight skeleton algorithm. This algorithm takes the outline of the building and automatically generates a plausible roof on top of it. While the result is often suitable for simple buildings, the spectrum of different roofs that can be generated is limited (see Figure 3.1a).

Some historic cities have a very complicated roof landscape that is divided into small pieces, whose roofs of individual houses do not necessarily correspond with those generated by the straight skeleton algorithm (see Figure 3.1b). Still, these roofs should be synthesize-able from simple parts. Different parts of a floor plan of a house usually have corresponding roof parts. These roof parts are often simple and consist of planar faces. The problem is that a simple nesting of the

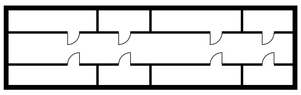
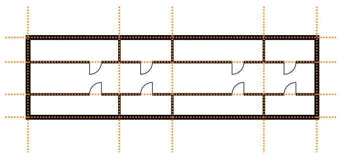
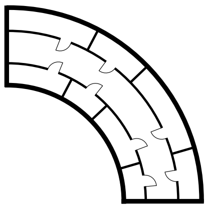
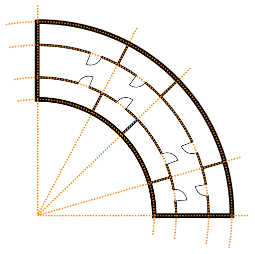
Euclidean geometry Many structures follow an euclidean coordinate system where walls follow construction lines (orange) parallel to the euclidean axes.		
Circular geometry Circular structures often have walls that follow concentric circles or lines that form a fan (orange).		

Table 3.1.: Geometry that is laid out in rectangular fashion can be split using the common split grammar methods (top row). Circular geometry is hard to model with split grammars (bottom row). Therefore, if the coordinate system is changed, and the splits of elements in the split grammar behave in a circular orientation, these structures can be reproduced with split rules. Here, the examples in the top and bottom row can use the same split rules, only the coordinate system is changed from Cartesian to circular coordinates.

parts does not always yield the desired result. This is because the roof parts can influence each other.

In order to solve these problems it has to be investigated how different roof parts of historic houses usually influence each other, and how these parts can be joined such that the correct geometry is achieved. The goal is the development of a novel framework for roof modeling which uses new high level union operators that automatically solve the possible geometric problems that can arise.

3.2.2. Round Building Geometry

Many historical landmarks have parts with elements arranged in a circular or spherical fashion. This might be towers (such as in Figure 1.2c), or domes (such as in Figures 1.2a, 1.2b, and 1.2e), or other structures such as rose windows or circular ornaments. There is usually a circular or spherical repetition of one or more elements that follows some architectonic rules.

The process of repetition in architecture is something that split grammars have already mastered, but they usually do it along a straight line. The method from

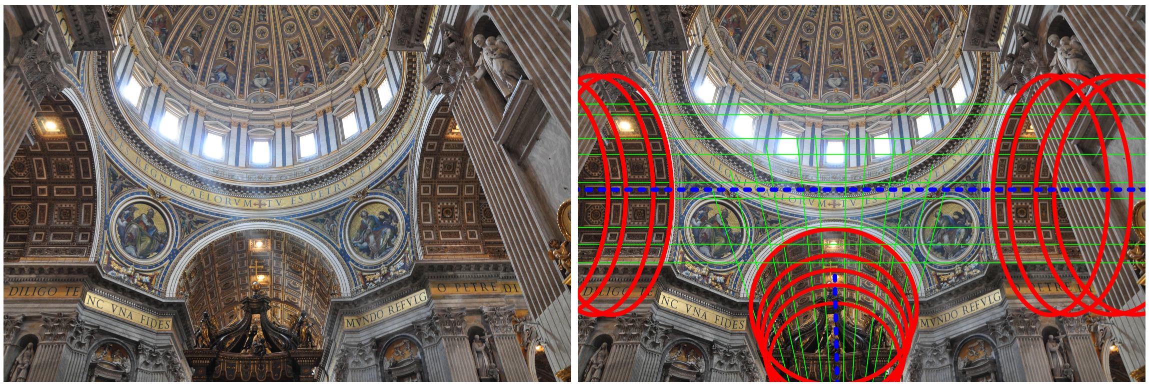


Figure 3.2.: Interior image of the St. Peters basilica (left). The elements at the arches are arranged circular and those at the dome spherical. The arrangements of these elements have their own reference coordinate systems. For the arches the two cylindrical coordinate systems are depicted (right). The blue lines depict the main axis. For the generation of the coffers, possible splits along the axis are depicted in red, and possible splits circular around the axis are depicted in green. Image taken from [1].

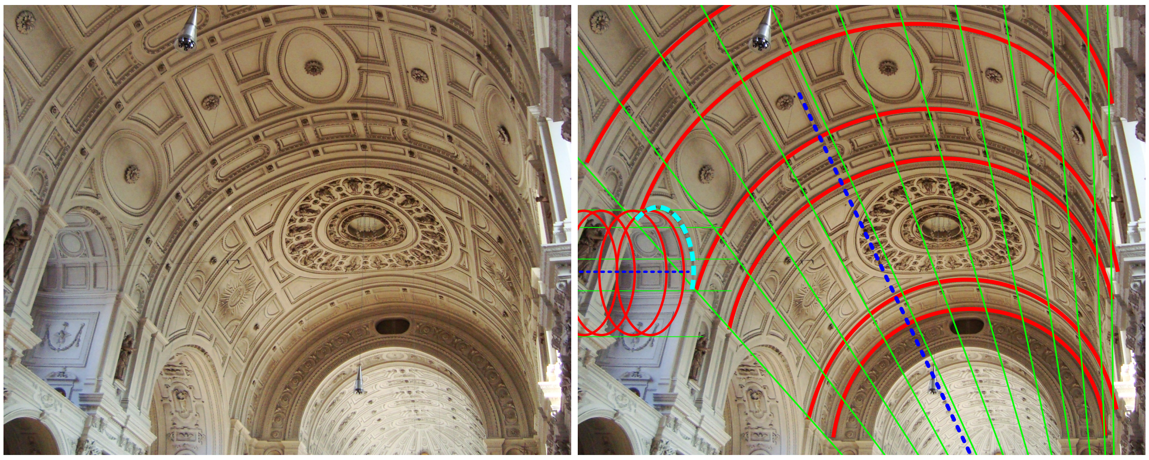


Figure 3.3.: Another example of arches of a church (left). Two possible setups for coordinate systems (right) are highlighted - there is the main one, and a smaller side one on the left. Colors are choosen as in Figure 3.2. The dashed light blue line depicts the line where both coordinate systems intersect. Having both coordinate systems available allows for the exact generation of the line and geometry on it. Image taken from [2].

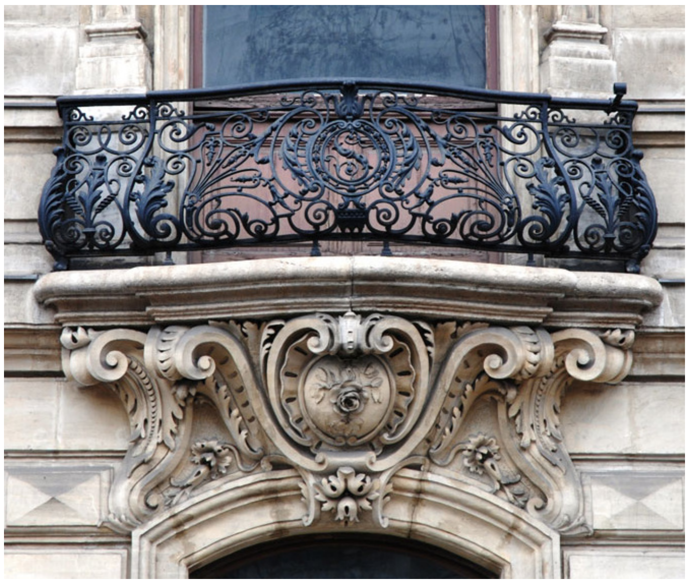


Figure 3.4.: Wrought iron balcony of the Palais Sturany in Vienna. The ornamental forms of such architectural elements are hard to model in the computer. Image taken from [3].

Section 2.2.6 is an exception since it produces deformed geometry, nevertheless the repetition is performed in a straight line and afterwards a deformation is applied.

There is certainly an opportunity to extend the common split grammar paradigm to work in a circular and spherical way. The idea is to create a new split grammar system, where the underlying coordinate system for different parts of a building is changed to cylindrical or spherical, while the common and proven split operators are adopted mostly unchanged (see Table 3.1 and Figures 3.2 and 3.3). That way, elements are arranged naturally in the required form.

3.2.3. Ornamental Forms

Historic buildings of certain architectural styles often have elaborate ornamental elements (see Figure 3.4). These ornamental forms are often hard to model in the computer. In fact, many modern buildings are completely void of ornamental forms. This might be an expression of modern architectural styles, but partially this might also be the case because modern buildings are usually planned with software which has only limited support for these kinds of ornamental forms.

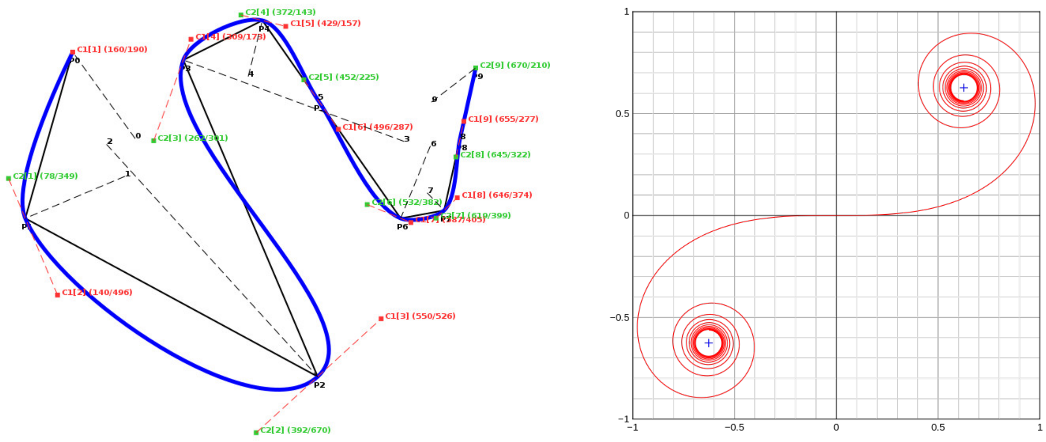


Figure 3.5.: Commonly used bsplines as curve representation have unfavorable curvature properties for many ornamental forms (left). Spirals can reproduce these forms better (right). Images taken from [4] and [5].

In general, ornamental forms consist of artistically arranged curves. The most common curve representation these days are bspline curves (see Figure 3.5). Bsplines have some problems for representing ornamental forms because their curvature properties are not optimal. The curvature of the curve is generally not as smooth as many ornamental forms. Spirals can model the circled forms of ornaments better.

A spiral based curve representation should be developed that is suitable for modeling of ornamental forms. For this, the usage and editability of curve control points has to be investigated, and a suiting user interface for modifying the curve is needed.

3.3. Guiding Hypotheses

In the beginning of this thesis (Section 1.1) the question was posed:

What can be novel techniques that allow complex historical architecture to be procedurally modeled in the computer?

The observations and considerations in this chapter can be summarized in the following research hypotheses:

- **H1**: *Historic buildings cannot be reasonably modeled with existing procedural shape modeling techniques.*
- **H2**: *Domain Specific Languages (DSLs) are well suited for the extension of procedural shape modeling techniques to model historic buildings.*
- **H3**: *The three most effective extensions for historical buildings are:*
- **H3.1**: *Roof landscapes*
- **H3.2**: *Round building geometry*
- **H3.3**: *Free form curves*
- **H4**: *The DSLs for the extensions can all be formulated using a common underlying formalism.*

The motivation for the last hypothesis **H4** will be given in the following chapter, which introduces DSLs as a very helpful concept for reaching the goals of this thesis.

Later, the achievement of these hypotheses will be validated.

4. Domain Specific Languages

Domain Specific languages (DSLs) have been around for a long time. In Lisp it was already very common to create embedded DSLs and solve parts of problems with the created DSL. A DSL allows for expressing problems in a particular domain in a concise and different way compared to traditional programing. This allows for better communication with domain experts by using the DSL codebase as communication tool with the domain experts, or might even allow the domain experts to program the system themselves.

While generic solutions in software engineering can cover a wide range of problems, they might be suboptimal for specific problems. Specific solutions can be tailored to specific problems and therefore yield better results.

There are three categories of specific solutions in software engineering as the paper [90] shows. The *subroutine library* approach packages reusable domain-knowledge as subroutines for a general purpose programming language. *Object-oriented frameworks* are an extension. While in subroutine libraries, the application invokes the libraries, in Object-oriented frameworks, the framework invokes methods from the application code. A *DSL* finally, is most tailored to the domain. It is usually small, declarative, expressive, and very focused. DSLs try to be very expressive and find the appropriate notations and abstractions for a given problem domain. They are therefore also called 'micro-languages', or 'small languages'. Because of their often declarative approach they can be a kind of specification language and be related to end-user programming (for example Excel macros).

The following is an introduction to Domain Specific languages and is mostly based on the book *Domain-specific languages* [33] from Martin Fowler.

4.1. Domain Specific Languages in general

There are two main reasons for Domain specific languages:

- Developer productivity
- Communication with domain experts

A DSL should be designed to have a very narrow scope and capture precisely the semantics of a domain, nothing more and nothing less. If it does, it gives a more natural way to express solutions than a general purpose language.

General purpose languages and DSLs differ in the way they model abstractions: In a general purpose language one has general abstractions like abstract datatypes, higher order functions and procedures, modules, classes, objects, monads, and many more that one can use to model the problem. A DSL in contrast should be the perfectly suited individual abstraction for the problem itself.

A DSL encodes the domain knowledge in a special human-readable form. It is still a computer language, meaning it is intended to be read and processed by computers, and must therefore fulfill certain requirements like being unambiguously defined. However, the domain knowledge is not coded in a general programming language (being hard to understand for non-programmers), or stored in an arcane file format.

Important for the design of a DSL is that there are sound abstractions. They should be easy to understand, highly modular, and straightforward to evolve. It is helpful to have an underlying semantic model that represents the abstractions. The semantic model can be something like an object model that represents how a system in the domain works. The DSL then represents code in a form that is close to the semantic model and the processing of the DSL code populates the semantic model (see Figure 4.1).

There exist different computational models that can be used for a semantic model. These can be for example:

- **State machines** (Figure 4.2): They have different states and translations between these states (such as a Turing machine). This lets one think about the process of the program differently than in common imperative programing languages.

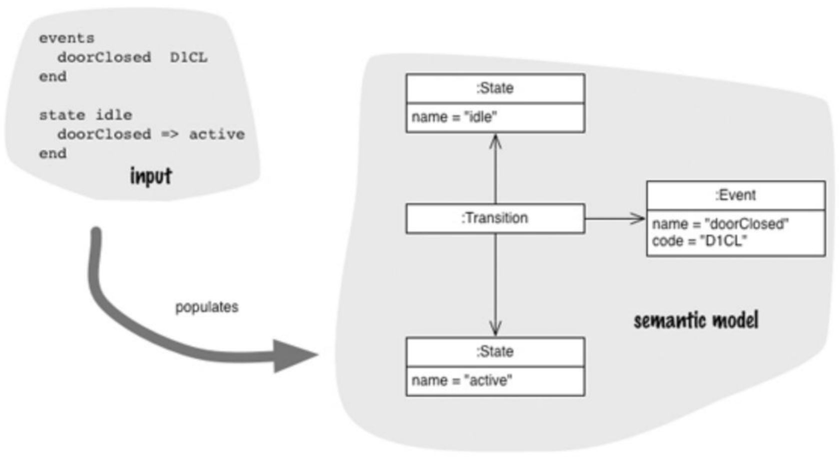


Figure 4.1.: Semantic Model (image taken from [33]).

- **Production rule systems** (Figure 4.3): They have textual rules with conditions and actions, where if the condition of one rule is met, its action is executed. These systems are especially useful for procedural modeling as they are the basis of shape grammars.

- **Decision tables** (Figure 4.4): They are similar to a production rule system, however their conditions and actions are arranged in a tabular form. This creates a more restricted system, which is simpler and more clearly laid out, and can therefore be better suited for domain experts.

- **Dependency networks** (Figure 4.5): They are models that describe dependencies between entities and are used for example in build tools such as `make`.

4.2. Comparison to General Purpose Languages

4.2.1. Advantages of DSLs over General Purpose Languages

Just like higher-level languages have some advantages over lower-level languages because they operate on a different level of abstraction, DSLs have some advantages over general purpose languages. DSLs are usually more concise and easier

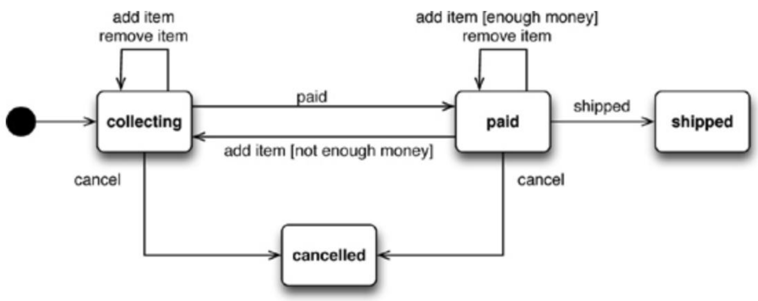
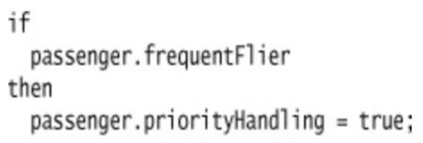
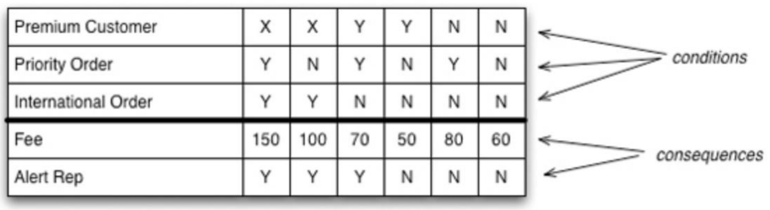


Figure 4.2.: State Machine (image taken from [33]).



```
if
  passenger.frequentFlier
then
  passenger.priorityHandling = true;
```

Figure 4.3.: Production Rule System (image taken from [33]).



Premium Customer	X	X	Y	Y	N	N
Priority Order	Y	N	Y	N	Y	N
International Order	Y	Y	N	N	N	N
Fee	150	100	70	50	80	60
Alert Rep	Y	Y	Y	N	N	N

conditions
consequences

Figure 4.4.: Decision Table (image taken from [33]).

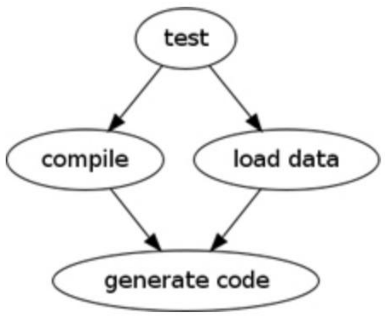


Figure 4.5.: Dependency Network (image taken from [33]).

to read. This makes them most notably easier to reason about and the intent of the program becomes more clear, which might be the most important property because complexity is often the biggest problem in modern software systems. Maintenance is also often a big component in modern software development and becomes easier too.

DSLs are usually simple to write, since they have only the syntactic elements that a domain requires. This might also allow domain experts themselves to write the DSL code, which further bridges the gap between developers and users. DSLs should usually be relatively restrictive and only allow necessary expressions. This makes them less error prone (with the possibility of having an additional type checking system), and less to learn (especially in external DSLs).

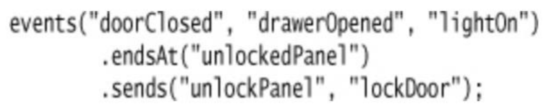
Finally, when the processing of DSL code uses advanced optimization techniques, the performance of the DSL code can in some cases be better than that of normal code.

4.2.2. Disadvantages of DSLs compared to General Purpose Languages

While one main advantage of DSLs is that they make the intent of a program easy to see, the behavior of the program is often harder to see. The behavior is to some extent intentionally hidden, which reduces the complexity of the code and makes it easier to read and reason about. The disadvantage is, that certain behavior can be unexpected and hard to reconstruct.

For a DSL there is much more work and costs upfront. The DSL has to be designed and the design then has to be implemented. When new users are using the DSL they have to learn the syntax and how it works. For people that are not domain experts, the DSL can be hard to understand in the beginning. However, at least the underlying problem domain they would have to learn anyway, independent of how it is encoded.

One critical thing is how a DSL is integrated into the software development process. DSLs are not widely used in the industry and therefore there does not exist that much design experience, there are less guidelines and design patterns, and less literature exists. Often the software creation process has to be modified, including design, implementation, debugging, and maintenance of the DSLs and their connections with the rest of the system. Especially in external DSLs



```
events("doorClosed", "drawerOpened", "lightOn")
        .endsAt("unlockedPanel")
        .sends("unlockPanel", "lockDoor");
```

Figure 4.6.: Internal DSL (image taken from [33]).

many convenient or even needed features, such as IDE support (e.g., syntax highlighting, code checking, etc.) are often not available or only available as generic DSL builder support.

Developing the design of a DSL can have some pitfalls. The scope of a DSL should be as small as possible, otherwise using a general purpose language would be better. However, often more and more features are added after time, which bloats the DSL definition and makes it harder and harder to maintain. For example, this could be argued as a disadvantage of the approach of CGA++ (described in Section 2.2.4). When the DSL is already in use and familiar to the users, a reluctance to change it can set in. This can result in trying to adapt the way a problem is modeled to the DSL, instead of adapting the DSL to the way the problem should be modeled in the first place.

Finally, DSLs can also have poorer performance than regular code. When the DSL optimizations are not good enough, manually written regular code that is tailored and specialized for the concrete problem can be much faster.

4.3. Major Categories of DSLs

There are generally two types of DSLs - external ones and internal ones. While external DSLs have their own domain optimized syntax, they are generally laborious to define and implement. Internal DSLs in contrary, are built into an existing programming language and use and extend its syntax and functionality.

4.3.1. Internal DSLs

An internal DSL is a DSL that is embedded into a host language. The DSL code that a user writes is a valid host language code, but written in a certain style that emulates being a language of its own. It is so to say a stylized use of the host language. This is good for programmers because if they know the host language already, they do not have to learn the syntax of the DSL, and they can also use existing and familiar tools such as IDEs with all their functionality. If multiple internal DSLs are embedded into the same host language they also benefit from having one and the same familiar look and feel.

Internal DSLs are easier to implement than external ones, since they do not need to be build from scratch. Parsing, code generation, or interpretation is all already provided by the host language and its ecosystem.

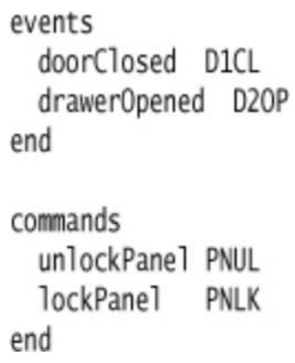
A big advantage is that general programming language features from the host language can be used, such as variables, conditionals, loops, functions, objects, etc. Mixing host language code and DSL code comes naturally. This helps to keep the scope of the DSL specification to the minimum that is needed for the individual problem domain, and general features can simply be modeled with the host language.

However, one has to be careful when designing the DSL not to be limited by the capabilities of the host language. Because an internal DSL is written in host language code it can also have a syntax that might not be ideal for the problem domain or simply look strange.

Internal DSLs are similar to simple libraries which are used in the host language. In fact, a simple library might be perfectly fine to use for modeling the problem domain. The difference is that an internal DSL has a more sophisticated syntax and interface which, for example, can make it look more like a human language and thus more read- and writable.

For the implementation of an internal DSL much of the work is already done by the existing parsers, compilers, interpreters, and tools of the host language. In special cases, a preprocessor might be used to transform DSL code to host language code. This can be done to add some syntactic sugar to the DSL and avoid distracting symbols or syntax from the host language in the DSL code (for example dots or braces that are not needed in the DSL code).

The build tools that are available for the host language allow for a rapid design of the internal DSL. They also facilitate change which is needed for experimentation, fault correction, and evolving the design, and reuse of artifacts such as syntax, semantics, implementation code, software tools, or documentation.



```
events
  doorClosed  D1CL
  drawerOpened  D2OP
end

commands
  unlockPanel PNUL
  lockPanel   PNLK
end
```

Figure 4.7.: External DSL (image taken from [33]).

4.3.2. External DSLs

In contrast to internal DSLs, external DSLs have their own individual syntax and need to be parsed and processed. So the up-front cost for developing them is much higher. However, once they are created, they usually have a nice and concise syntax for the problem domain, which is shaped to support the necessary expressions (and not more) and is not bound to follow the syntax of any host language. This is especially advantageous for domain experts, since they do not have to bother with the syntax of a host language that is probably unknown to them and has to be learned additionally. But also when the DSL is used by programmers, the simple syntax brings great advantages for the communication with domain experts.

For the implementation of an external DSL some existing tools for lexing and parsing are available (for example lex and yacc). The result then either needs to be interpreted, and an interpreter has to be written, or code for another language might be generated that is then run.

5. Case Study: Constructive Roofs from Solid Building Primitives

This chapter was published as the research paper *Constructive roofs from solid building primitives* [25] in *Transactions on Computational Science XXVI*. It was written at the *Institute of Computer Graphics and Knowledge Visualisation (CGV)* [46] at *Graz University of Technology* in collaboration with the *School of Computer Science and Engineering (SCSE)* [76] at *Nanyang Technological University, Singapore*. The paper has been adapted here.

5.1. Introduction

With the growing popularity of virtual 3D worlds in games and movies, various VR simulations and 3D street walk-throughs of todays cities or cultural heritage scenes, attention of researchers shifts towards quick and flexible ways of modeling or reconstructing large numbers of buildings.

While manually generated models typically yield the highest visual quality, the effort to produce such results is immense. The research addressed at this problem can be grouped into two approaches: automatic reconstruction from measurement data, and automatic generation using procedural modeling.

While automatic reconstruction might produce a faithful appearance of the original object, the result is often a (possibly dense) 3D mesh. This results in serious limitations to the modifiability of such models, as manual effort is required to change such models.

On the contrary, procedural modeling is an abstract representation of the building process of a model. Changing the procedural description typically requires less effort than manual 3D modeling, however the procedural system has to keep the balance between automation (less manual effort, but also less detail control

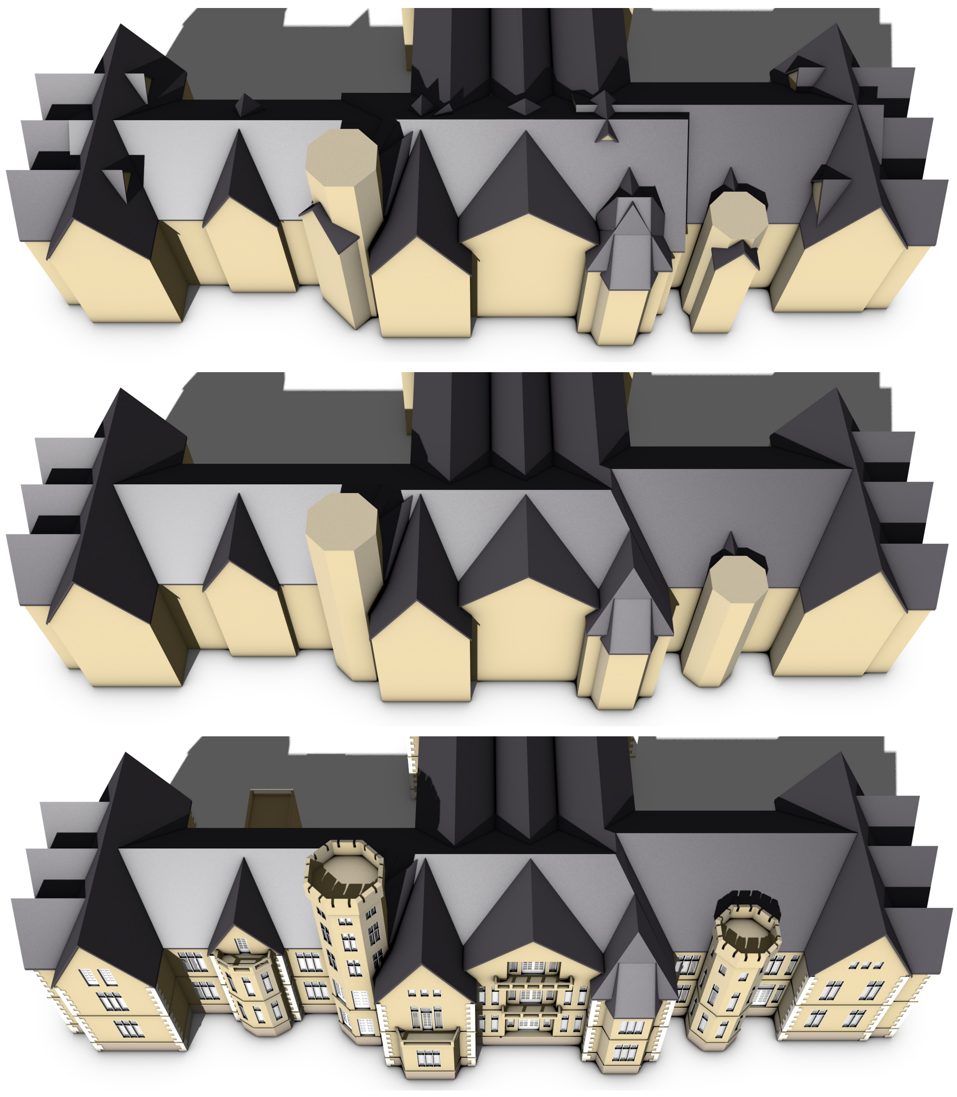
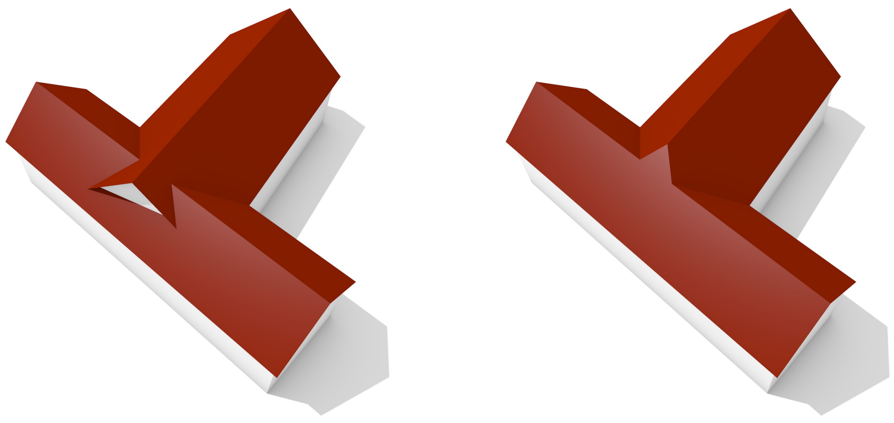


Figure 5.1.: Modeling a building by parametrizable parts greatly reduces the necessary effort. A simple union of parts might lead to undesirable results (top). Therefore, we introduce automatic roof *trimming* for solid building primtives (middle). The resulting model can be further refined using existing procedural approaches (bottom).



(a) Simple union (b) Trimmed version

Figure 5.2.: A simple union of building parts with roof does not produce a desirable result (a). The influence of parts can be non-local, in this example the bounding plane of the roof geometry of the smaller building trims the roof geometry of the higher building and induces a small triangular roof area (b).

over the result) and expressiveness of the system (which types of models can be created).

Current procedural modeling methods use mass modeling to define the coarse outline of a building by representing it via a number of parts. Rules further refine the geometry of these parts. Basic interaction between parts is possible via occlusion queries. Automatic roof synthesis can be done, but does not reflect all possible roof configurations present in real buildings, and the user has few if any possibilities to control the result of the roof generation process. In reality, this process is governed by rules, but it is not deterministic.

Especially buildings in historic cities can often be modeled by simple parts, but achieving the final roof geometry is not possible by simple nesting or boolean union of the parts. The problem here is that the roof of one part can be influenced by the roof of another part (see Fig. 5.1 and 5.2). Depending on the constellation of the parts, different topological connections arise (see Fig. 5.3). These problems can increase with growing irregularity of the building. Fig. 5.4 shows a view of the city of Graz, Austria. Merging and influencing roof parts form a complex roof landscape. Still, blending of roof faces follows some consistent rules which we formalize in this chapter.

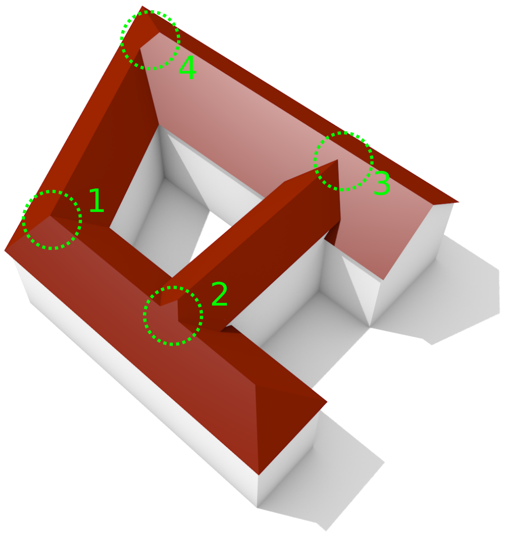


Figure 5.3.: Complex roof shapes. At (1,2,3,4) the geometry has different topologies, depending on how roof-parts merge together. At (1) four roof-faces meet in one vertex, which is a special constraint since in general four planes do not intersect in one point.

The goal of this chapter is to present a method that will allow representing a great variety of coarse building structures with roofs using a concise declarative approach. The structure is modeled by parts and their geometric influence to each other. The resulting structure can be input for a rule-based system that refines the geometry of the building parts. We show an example of both techniques combined in Fig. 5.1 and 5.21.

We make the following Research contributions in this chapter:

1. We present an *abstract building model* specification that facilitates a concise description of a building assembled of several parts.
2. We introduce a method for automatic geometric trimming of adjacent building parts that influence each other.

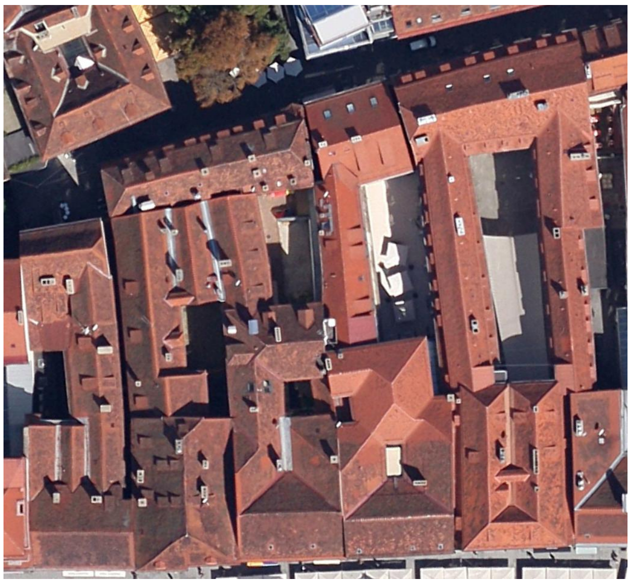
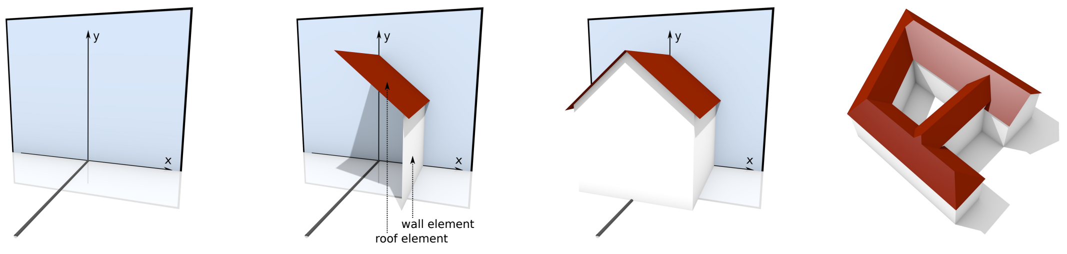


Figure 5.4.: Aerial image of the inner city of Graz, Austria. Different roof parts merge and form a complex roof landscape. There are many implicit dependencies of parameters of roof faces like slope, height of eave, etc. This roof landscape is not easily created with existing modeling tools. Imagery ©2015 Google, Map data ©2015 Google.

5.2. Related Work

Shape grammars have been introduced by Stiny et al. [79] in order to generate paintings using rule systems. In [96] the concept was extended to split grammars in order to model 3-dimensional structures, especially building facades. The concept was further refined by [65]. Several other works were built on this principles, extending it to interconnected structures [50], extended systems of rule application [51], or more general split rules [87]. These systems can use the aforementioned mass modeling approach to generate rough geometry. While most of these systems support basic roof generation, complex roof structures are either fully automatic (using a straight skeleton approach), limiting the number of possible roof structures, or the roof structures have to be modeled in detail for each part, resulting in a complex description.

3D construction of building roofs is an often covered topic in literature. Most papers are however concerned with reconstruction of roofs from image or scan data [49, 43, 83, 92, 11]. A good overview is given in [36]. The aforementioned methods usually use variants of plane fitting. The work of Milde et al. [64, 63]



(a) Axis (b) Side (c) Solid (d) Structure

Figure 5.5.: The components of our abstract building model. An *axis* is given for orientation. Relative to an axis, a *side* is defined. A side has wall and roof elements. Multiple sides form a *solid*, which is the basic building block for a *structure*.

and Dorschlag et al. [23] use additional grammar and graph based approaches to further aid the reconstruction process. The method presented by Fischer et al. [32] uses an approach with connectors to align the extracted planes.

While they can produce good results, the output is often only a polygon mesh without semantic information. This is not very suitable for scenarios where a modifiable model is needed (e.g., urban planning), which is why recent methods use primitive fitting (with simple convex houses and gabled, hipped, flat, etc. roofs) where the primitives could be seen as semantic units [44, 45].

When models of non-existing objects have to be built (e.g., for movies, video games, etc.) none of the mentioned methods is applicable. There are many papers which deal with the 3D construction of buildings, however, the roof is most of the time only a small part of the solution and often modeled in a primitive and simple way. For example,

- [65, 30, 29, 56, 81, 82] use simple roof-primitives with often convex ground shape plans and gabled, hipped, flat, etc. roofs, and combine them to generate their roofs,
- [84] has a special specification for Asian roofs,
- [62] and [52] use the famous straight skeleton algorithm [9] to generate the roof.

The methods with simple roof-primitives can yield good results for very regular buildings, but when the roof gets more complex and irregular, these methods are not able to fully reproduce the shape. Special specifications, like in the case for

Asian roofs, are very suitable for the domain, but lack the possibility to describe a broad spectrum of roofs.

The straight skeleton algorithm is very suitable for generating general roofs, as its resulting skeleton corresponds to the edges of a roof on the building. Using it, a roof on an arbitrary simple ground polygon can be created. The downside is that the generated roof is only one of many possible roofs, and there is no way to get another roof. Here, an extension to the straight skeleton, called the weighted straight skeleton [13], brings greater flexibility. The work of Kelly et al. [48] uses the weighted straight skeleton to model not only arbitrary roof-shapes, but also the whole outer shell of a building. One drawback of this method is that straight skeleton algorithms (even more in the weighted case) are hard to implement due to algorithmic problems when numerical errors arise.

The problem of roof modeling also arises in geographic information systems. Open street map [67] is a project that lets volunteers all over the world collaborate in creating mapping data for streets, buildings, borders, etc. In order to model roofs, they have built a large categorization table of roofs. Each category has its own set of parameters. So a large class of buildings can be declaratively modeled by specifying the category and the corresponding parameters. Also, the straight skeleton method is contained in the categorization. Additionally, a method proposed for open street map is roof-line modeling. Here, the modeler traces roof lines on an areal image (from top-down view) of the building and provides additional information, e.g. a type (ridge, edge, apex, ...), or height.

What is mostly missing is the accurate handling of merging roof faces of neighboring houses. In historic cities, houses that are adjacent often share the same roof plane. Depending on how the houses connect, the roof parts merge in different ways. While automatic reconstruction via plane fitting might reproduce the situation correctly (depending how good the algorithm is), methods that work with mass modeling primitives have difficulties, because for the merging the primitives must be changed. The weighted straight skeleton is capable of modeling these connections, but it might become unhandy when big roof regions with multiple roof part connections are modeled, because the whole roof must be modeled as one piece to ensure consistent geometry at interconnections.

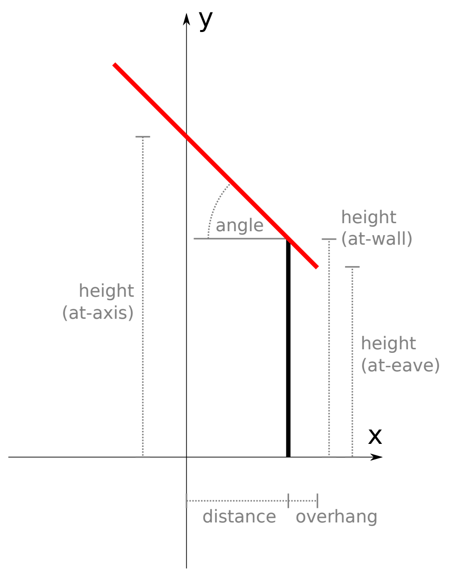


Figure 5.6.: Cross section view of a side. A side in our abstract building model consists of wall and roof elements. Here we show one possible parameterization (parameters in gray) from which the wall (black) and roof (red) elements are derived.

5.3. Solid Building Primitives

In this section we give an overview of the building abstraction, and specify the individual components of our system.

The core abstraction in our system is the assembling of a building by separately defined parts, which are called *solids*. As most buildings are composed of planar walls, each solid is composed of several planar *side* parts, which represent a cross-sectional profile of the corresponding wall and roof part. Each side part corresponds to a line segment of the ground polygon of a solid. The geometry of a solid is obtained by intersection of half-spaces that correspond to the profile line segments of each side part.

A simple union of solids defined in this way might lead to undesirable results (see Fig. 5.2). However, we have observed that in many situations the correct result can be obtained by trimming solids by corresponding half-spaces of side parts of adjacent solids.

5.3.1. Basic Components

Our abstract building model is composed of several components. We will now give a explanation of the basic components in a bottom-up manner (see Fig. 5.5):

Axis
: is a directed line segment in the ground plane, defined by a start- and an endpoint.

Side
: corresponds to a planar building part consisting of a wall and a roof element. These elements are specified as line segments of the cross-sectional profile of the part, with respect to a reference axis that corresponds to the direction of a line segment of the ground polygon. Various parameters define the shape of the side (see Fig. 5.6).

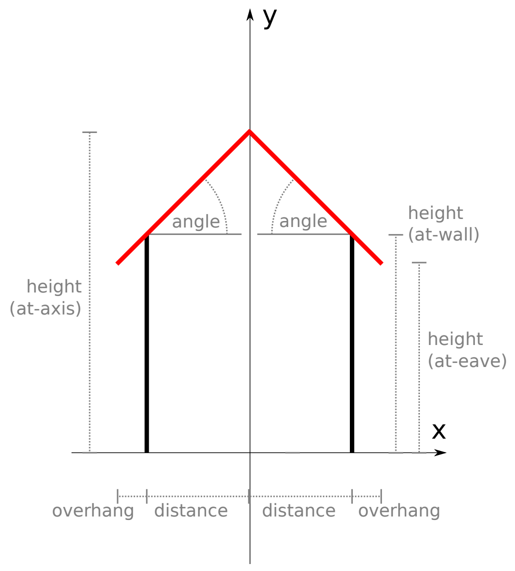
Solid
: is composed of multiple side components whose wall elements correspond to a convex ground polygon. Each line segment of the cross-sectional profile of a side corresponds to a half-space in 3D.

Structure
: is the combination (grouping) of several solids. Structures can be nested, e.g. for representing dormers.

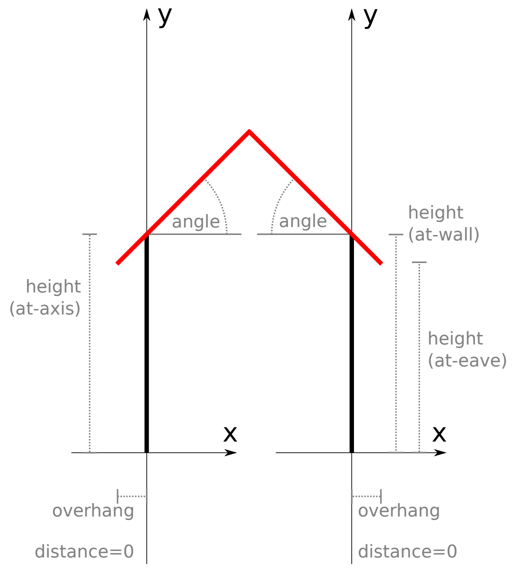
5.3.2. Parametrized Sides and Solids

Our definition of the side component allows to model a rich variation of different roof types and building parts. It is however relatively low-level, therefore we introduce parametrizations for sides and solids, that allow us to reduce the necessary effort to describe a building that is composed of similar parts. For example, if a building is modeled from areal photographs, it is practical to define sides relative to ridge lines of roofs. When using ground polygons from a GIS (geographic information system) database, sides should be relative to the ground polygon. Different parametrizations for sides can be seen in Fig. 5.7.

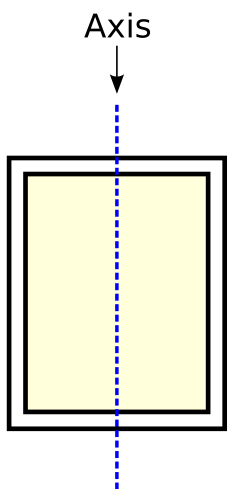
The parametrization includes an *overhang*, as roofs often extend over walls in order to prevent rain falling on walls.



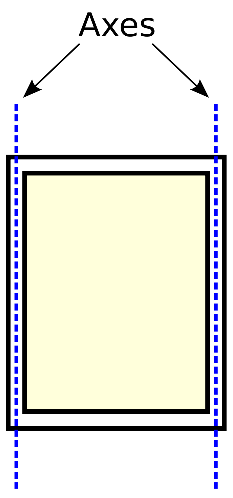
(a) Side parameters defined relative to one axis.



(b) Side parameters defined relative to two axes.

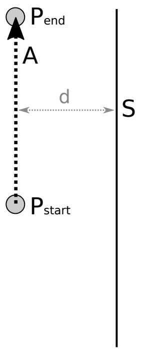


(c) Floor plan

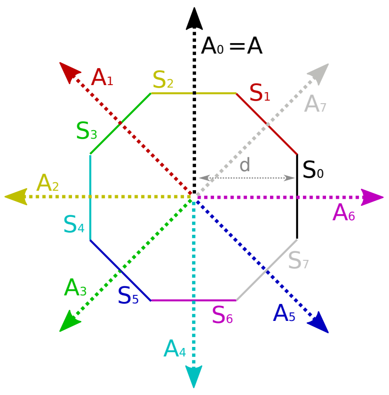


(d) Floor plan

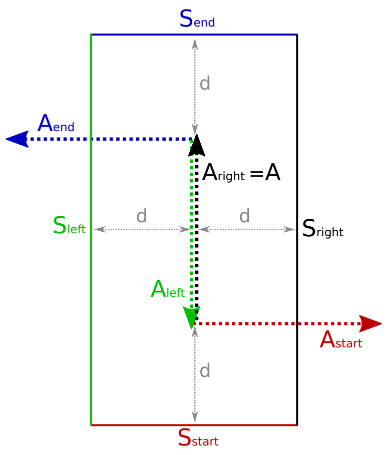
Figure 5.7.: A solid is composed of multiple *side* components which can be parametrized arbitrarily. In (a,c) *side* components are specified with respect to the ridge (both *side* components use the same reference-axis and coordinate-system). (b,d) *side* components are specified with respect to the walls (*side* components use different reference-axes and coordinate-systems).



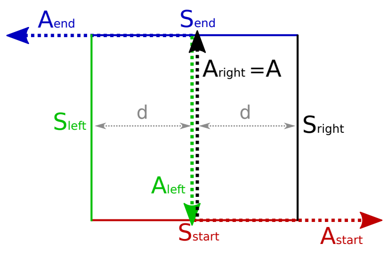
(a) Axis defined by start- and endpoint and its side.



(b) Generation of an octagonal footprint.



(c) Generation of a rectangular footprint. The parameters for each side are copied.



(d) Generation of a rectangular footprint. The parameters for start and end and for left and right are copied differently.

Figure 5.8.: Automatic generation of sides for solids. Based on an axis A, given by two points P_{start} and P_{end} and its according side S (a) additional sides for a complete solid can be generated. The axis are denoted A_i and the according sides S_i. Each side has parameters, the distance is here denoted with d. Different copy mechanisms allow for shapes in rectangular (c,d) or circular arrangement (b). The original parameters of the side are either copied for all parts (b,c) or modified while copying (d).

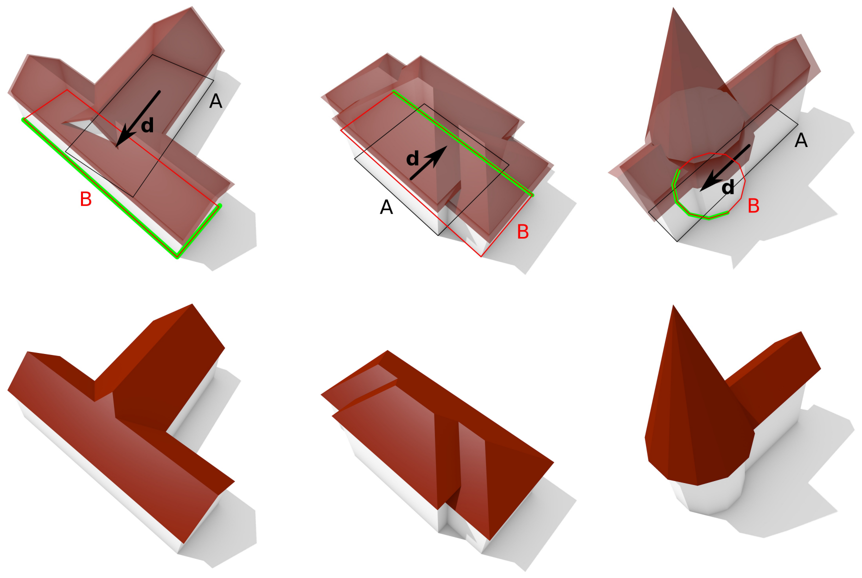


Figure 5.9.: We demonstrate the concept of trimming for solids by interactions of different building parts (top row). The user specifies an influence direction for each connecting solid (black arrow), the system automatically identifies the sides which will influence the result geometrically (green lines on the ground polygon). Only sides corresponding to green segments will be used for the trimming (bottom row).

5.3.3. Automatic Generation of Sides for Solids

As described, the user can specify axis in the system. This can be done i.e. by tracing of aerial images. Subsequently a side is assigned to this axis. In order to reduce the amount of work needed, all sides of a solid can automatically be generated from this one given side. In the standard case of a rectangular solid this would be four sides generated from one given side, but also n-gon constellations are possible (see Fig. 5.8).

The parameters from the original side are automatically copied to the newly generated sides. But this process can also be modified. For example, when the ridge of a hipped house is traced, the copying of all parameters is useful since the roof has the same properties on all sides (slope, eave height, etc.) (see Fig. 5.8c). However, for a gabled house the start and end sides are different (see Fig. 5.8c). Here we tell the generation process to change the parameters for the start and end sides. The distance is set to 0, and the roof components of this sides are removed.

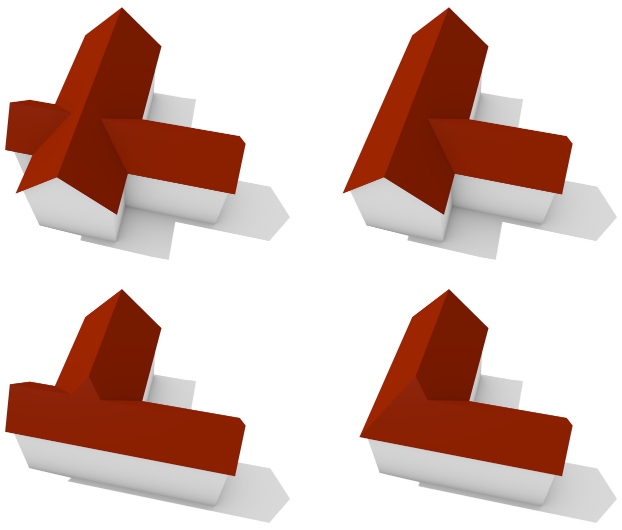


Figure 5.10.: Two connected solids A and B can influence each other in four possible ways: No trimming (top left), A trimming B (top right), B trimming A (bottom left), and both solids trimming each other (bottom right). Note that different combinations would be possible by changing the influence directions.

5.3.4. Automatic Trimming of Solids

When multiple *solid*s are combined using a simple union, unwanted configurations can occur like in Fig. 5.2a. Therefore, we introduce *trimming* between solids: the geometry of one solid should be truncated by parts of the geometry of another solid. In order to specify which parts get trimmed, a direction has to be specified.

Recall that each side of a solid consists of wall and roof elements, which correspond to half-spaces in 3D. Assume having two solids A and B. We specify that A gets trimmed by B according to the direction d (see Fig. 5.9). Then, for each side s of B there is a test whether the outward facing normal vector of the wall component of s and d form a positive scalar-product. If yes (green segments in Fig. 5.9), the wall sub-solid of A is intersected by the half-space formed by the wall component of s and the roof sub-solid is intersected by the half-space formed by the roof component of s.

A trimming connection is directed, as one solid gets trimmed by another. Therefore, two solids yield four different possibilities of trimming variations between

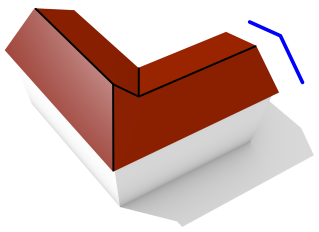


Figure 5.11.: A building that consists of two solids is evaluated with different parametrizations of the sides. In this example, only the slope of the roof of the right solid was gradually incremented from the first to the last image and the model was re-evaluated. It can be observed that the roof geometry adapts accordingly to the situation.

each other (see Fig. 5.10).

Using this method of trimming we can ensure that the influence between solids adapts correctly to changes in the parametrization. As an example, Fig. 5.11 shows the results of automatic trimming under varying roof angles for one solid.

5.3.5. Roof Shapes

Up to now, we assumed the roof element of a side to be a single planar element. However, Mansard roofs have two or more slopes on their sides. We account for this by introducing side parametrizations that generate more than one roof element, which is called *profile-polygon* component in our specification. We show examples of a few roofs with profile polygons in Fig. 5.12.

Automatic trimming is performed in the same manner, although the case in which neighboring solids influence each other in a way that introduces additional geometry (compare to the small triangular area that emerges in Fig 5.2a) might not be well defined. This, however, is not of practical importance, as we did not encounter such situations in real roof configurations, where these roofs usually have the same ridge height.

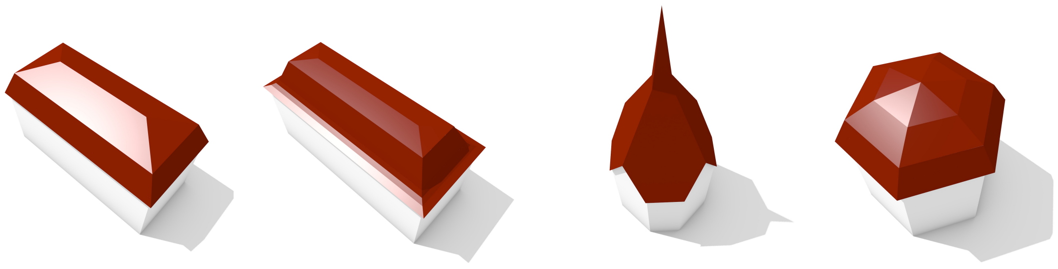


Figure 5.12.: Some roof shapes consist of more than one planar element, e.g. mansard roofs. We account for such types by a general profile-polygon type. The last two examples show solids with non-rectangular ground polygons.

5.4. Specification of the Building

In order to model a building we have to parameterize and combine multiple solids. We developed a specification with which this can be done in a structured fashion for one building.

5.4.1. Structure of Specification

Fig. 5.14 shows the elements of a structuring language we have developed. A building is represented as a structure, which consists of multiple solids. Each solid has at least one side, and a side has a parametrization (as already covered in Section 5.3.1).

To achieve the correct influence of solids as described in Section 5.3.4, a solid can contain multiple trimmings. A trimming references to the influencing solid and an axis that specifies the direction.

Additionally, a solid can also have dormers (see Fig. 5.13). A dormer is a structure, which is put on the solid and where appropriate cutting of roof faces is done on the solid. For repetition of the same dormer multiple times along an axis, we provide the dormer-pattern construct, where a repetition pattern must be provided. These patterns are formed in analogy to repetition or subdivision patterns in shape grammars.

This specification can easily be extended for further building and roof elements (e.g. chimneys, ..) or different side parametrization.

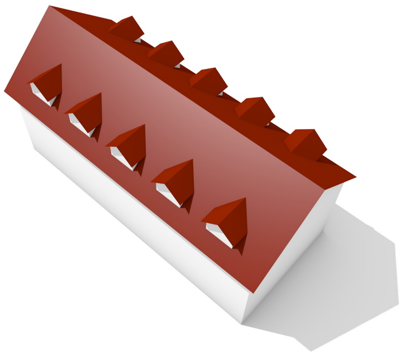


Figure 5.13.: Dormers on a solid. They are structures which are placed on a solid, either individually or according to a pattern.

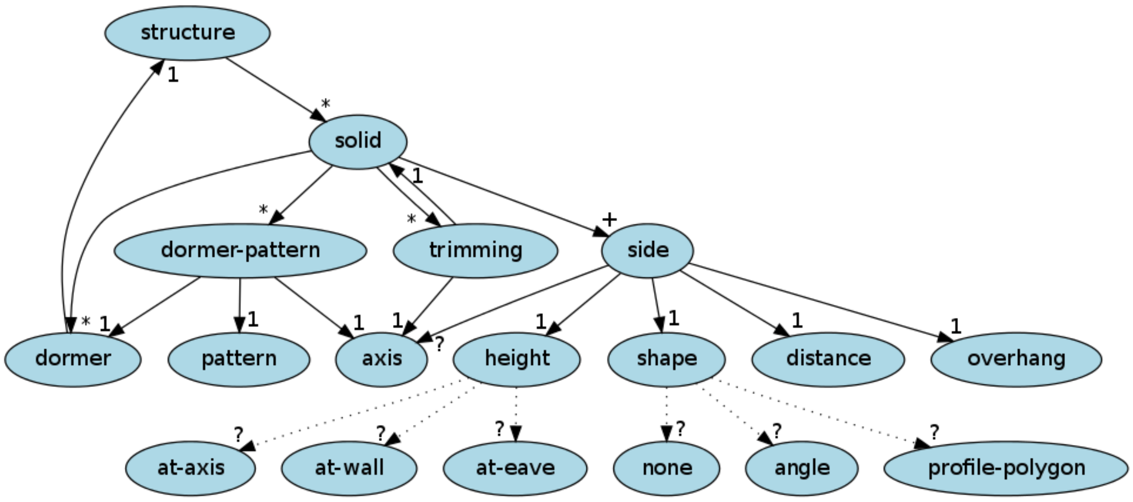


Figure 5.14.: Our abstract building model: The coarse structure is specified by solids and their influence to each other. Nodes of the graph show elements, arrows show their relationships (in a classical *has-a* notation). The quantifiers show the number of sub-elements (? ... zero or one, 1 ... one, * ... zero or multiple, + ... one or multiple).

5.4.2. Geometry Operators

The geometry of roofs must often fulfill certain criteria. Important properties for the roof can be for example:

- Faces are planar (all vertices of a face must be co-planar - this can be a problem when a face has more than three vertices).
- Faces intersect in one common vertex (for more than three faces this can be a problem).
- Ridges are horizontal (often means that the to the ridge adjacent building sides must be exactly parallel).
- Faces merge over multiple building parts (this often means that the corresponding walls must be co-linear).

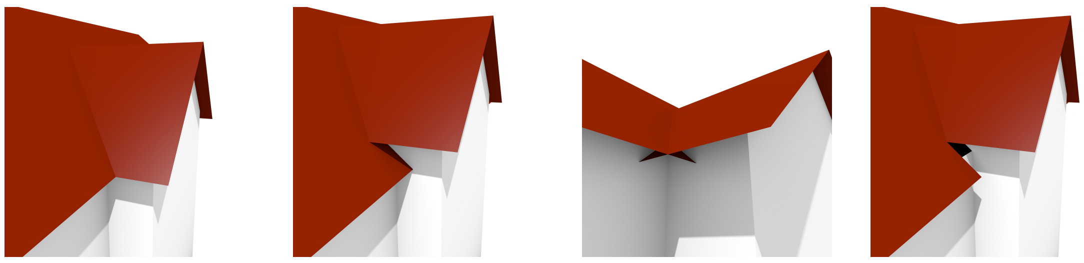
These properties form parameter dependencies, and they are a problem for the input format, since there must be a way to avoid over-specification of parameters (and therefore violation of the dependencies). Because of that, we included geometry operators in the language. They operate on the axis element and can produce new axis elements that allow for fulfilling the requirements stated above. The operators are for example translation, rotation, parallel-translation, and line-intersection.

5.5. Implementation

5.5.1. Double-covered Areas

The geometry of a roof can be described as a 2-dimensional manifold where every possible vertical ray intersects the roof exactly 0 or 1 times. This means that the roof geometry can be generated as the surface of solid geometry. The idea is to generate solid geometry (which easily allows boolean operations) and then extract the roof faces from it. The exception are double-covered areas (see Fig. 5.15). Here the roof of one part of the house rises over the roof of another part (see Fig. 5.15b), and a vertical ray can intersect the overall roof multiple times, which complicates the algorithm.

At a double-covered area, the lower part of the roof must extend to the wall of the other solid. But when the eaves of both solids are at the same height (see Fig.



(a) Eaves at same height. No area of the roof is a double covered area.
(b) Eaves at different height. A double covered area is formed.
(c) Same height - wrong result if the roof geometry extends to the adjacent wall.
(d) Different height - wrong result if the roof geometry does not extends to the adjacent wall.

Figure 5.15.: Eaves and double-covered areas. Eaves are a challenge for the geometry generation. Eaves with the same height (a) need a different geometry generation than eaves with different heights (b). We show two examples (c,d) where simple algorithms yield wrong results.

5.15a), they are not allowed to extend to the other solids wall, otherwise incorrect geometry on the bottom side of the roof would occur (see Fig. 5.15c). In fact, only the roof part with the lower eave is allowed to extend to the other solids wall.

5.5.2. Organization of the Components

In order to extract the geometry we generate convex polyhedra for each side and perform boolean operations on them. Fig. 5.16 shows the *inner* and *outer side geometry* for various sides. They are both identical, except that the outer one extends the amount of overhang farther outward from the wall. Through this, we will later be able to generate the geometry for overhangs and double-covered areas.

The geometry for a solid is generated by intersection of the geometry of all sides. Taking either the inner or outer side geometry, this yields the *inner* and *outer sub-solids* as shown in Fig. 5.17. The inner sub-solid is used for the extraction of the geometry for the walls, and the outer sub-solid is used for the extraction of the geometry of the roof. An exception are roofs with double-covered ares. Here the roof of one solid is covered by the roof of another solid. So for each side of a solid, depending on the situation, the inner or outer side geometry needs to be taken. Then an individual sub-solid is formed by the intersection of the side geometries.

5.5.3. Trimming

As already described, a solid can be trimmed by another solid. Which sides of the other solid are selected for the trimming is described in Section 5.3.4. For the geometry extraction this means that the selected sides of the other solid are simply added as sides to the trimmed solid. This way, the solid gets automatically cut because its geometry is determined by the intersection of the geometry of its sides.

5.5.4. Geometry Extraction

The geometry of the walls can be obtained by performing a boolean union of the inner sub-solids (see Fig. 5.17) of all solids, after they are trimmed. Then the sidewards facing face of the resulting volume form the walls of the structure.

The geometry of the roof could also be obtained by performing a boolean union of the outer sub-solids of all solids. Taking all the upward facing faces of the resulting volume would yield the roof geometry. But double-covered areas make the geometry extraction for the roof more complicated. This method would lead to geometry errors (as seen in Fig. 5.15d).

Therefore, for every solid the roof geometry is extracted separately. The algorithm for extraction is shown in Algorithm 1 and the according description is presented in Table A.1.

The result of the geometry extraction can be seen in Fig. 5.18. Note that the roof has the correct geometry, also in the inside of the building (the roofs bottom side).

5.5.5. Processing

We have built a system with multiple steps where the input data is transformed. The input format is XML but we built our own short notation that is semantically equivalent but better read- and writable than pure XML.

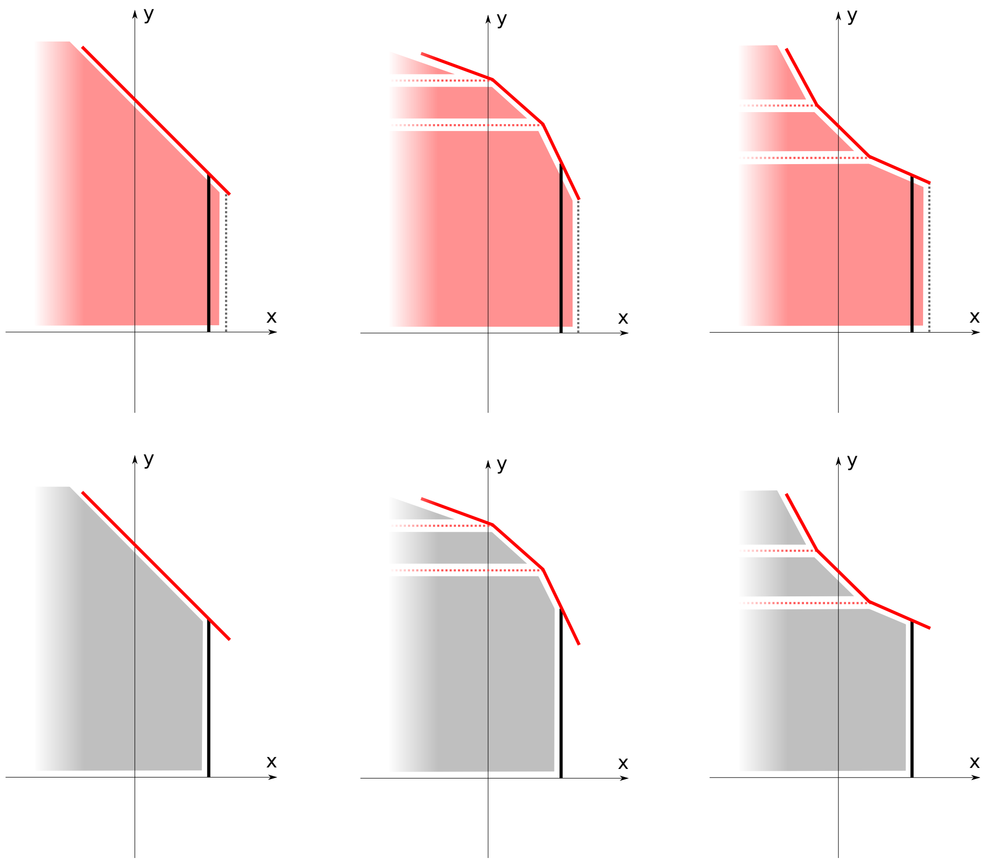


Figure 5.16.: Side geometry. For the geometry generation each side is represented via convex polyhedra (shaded areas). Top: *inner side-geometry*. Bottom: *outer side-geometry*. First column: Simple roof with one roof face per side. Second column: The roof is specified via a convex profile-polygon. Third column: The roof is specified via a concave profile-polygon.

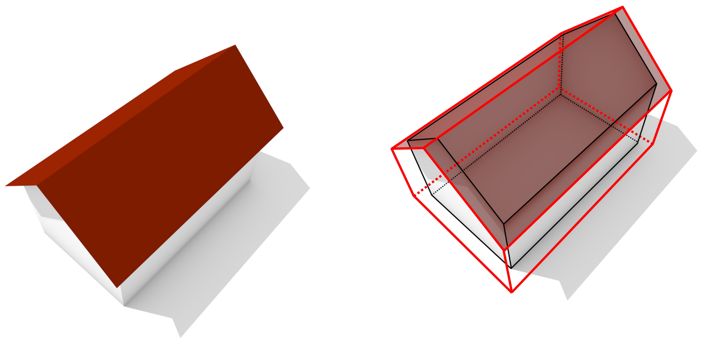
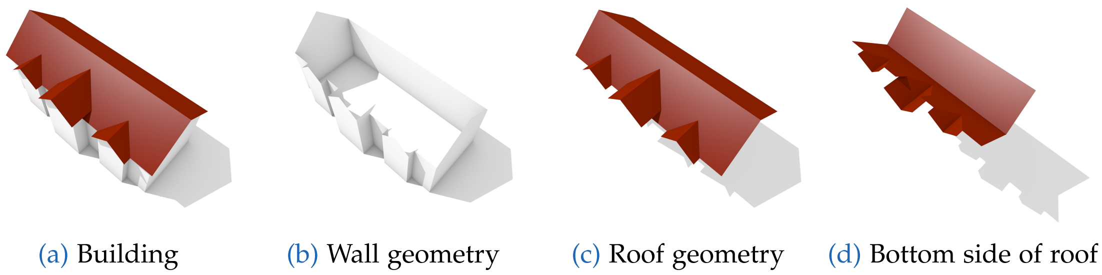


Figure 5.17.: Each solid consists of two sub-solids, a wall sub-solid (black outline) and a roof sub-solid (red outline), which are utilized for the trimming process.



(a) Building (b) Wall geometry (c) Roof geometry (d) Bottom side of roof

Figure 5.18.: Extracted geometry of a building. Wall and roof geometry are extracted separately. The bottom side of the roof shows that the solids are not simply sticked together, and the geometry is extracted correctly.

The steps of our current implementation are as follows:

1. The input file in short notation created by the user is converted into XML.
2. The XML file undergoes an enrichment step where sides and parameters are automatically added according to Section 5.3.3.
3. The new XML file is converted into GML (our own in-house programming language for procedural modeling).
4. Our engine in GML creates the geometry according to the principles described in Section 5.5.

5.6. Results and Applications

We evaluated our prototype implementation by modeling three scenes that exhibit interesting roof structures from aerial images. Our workflow consists of the following steps:

1. Trace important lines via a vector-drawing program.
2. Convert lines into axis.
3. Assign parametrized sides with respect to axis.
4. Assign sides to solids.
5. Specify trimmings between solids.

The result is an abstract building description in an XML file that reflects our specification (see Fig. 5.14). As of now, the workflow includes some manual steps, for example, the assignment of parametrized sides. We observed that while the models have complex roof cases, the roof shapes follow simple geometric rules when blended together. As it was expected, the decomposition into convex parts turned out to be quite obvious. Evaluation times were measured on the CPU implementation of our prototype system on a 2.6GHz quad core machine.

The first model is the royal palace of Milan (Fig. 5.19) where we also placed some dormers on the roof. The model consists of 28 solids (excluding dormers), and its evaluation took 6 seconds.

The second model is a part of the city of Graz, Austria (Fig. 5.20). It consist of 7 structures that have 34 solids in total, and its evaluation took 5 seconds. The roofs have considerable irregularities, which became a challenge in the modeling process.

The third model is the Magdalena palacio in Santander (Fig. 5.1 and 5.21), which is assembled from 25 solids, and its evaluation took 8 seconds without the facade detail and 20 seconds with the facade detail. The roof of this building exhibits some interesting features, such as the cavities on the middle part of the building, which are hard to model using fully automatic (e.g., straight skeleton based) approaches.

When doing direct comparison, the third model exhibits a more constrained structure: the ridges follow two principle axes which are perpendicular to each other. So we realigned the traced lines from the aerial view with the operators described in Section 5.4.2. A more irregular layout, like that in the first and second model, can make the modeling harder; at regions where solids adjoin or intersect, artifacts are more likely to arise.

For all models, we did not have any measurements, therefore we estimated values for missing parameters, e.g., the side heights, by using photographs of the buildings as guidelines. While in this case the result is not a fully accurate model, it produces a good approximation and visually satisfying result (e.g., see Fig. 5.21). Due to the missing parameters, we had to try different parameter sets several times to ensure that the roof has the right topology. Nevertheless, our abstract definition allows for quick reparametrization if real measurements become available.

The geometry can be exported to .obj or other file formats for further usage in other programs. We did our renderings in Blender where materials and lighting were defined.

5.7. Conclusion

Modeling the coarse structure of a building by the composition of parts proved to be suitable for buildings of classical architectural style. However, in many cases a building is not a simple union of such parts. In this chapter, we proposed automatic trimming of adjacent parts, which facilitates a concise, abstract decomposition of a building into parts and their geometric influence. A coarse 3D model of a building can then be generated from such a description, which can further be fed into a conventional pipeline (e.g. shape grammars) that generates detailed geometry for building parts (e.g. windows and doors), see Fig. 5.21. Our abstract building description facilitates easy re-parametrization of building parts, as the 3D model just has to be re-evaluated after a parameter has been changed (as seen in Fig. 5.11).

We see applications for our method in production pipelines of virtual worlds. This includes the digitalization of cultural heritage, since it can reproduce the complex geometry of roof landscapes in historic cities. It is also suitable for the movie and video games industry, where sometimes man years of work have to be spent on modeling the environment. Our approach, which integrates into existing procedural methods, further enhances the expressiveness of those methods and advances the automatic generation of geometry.

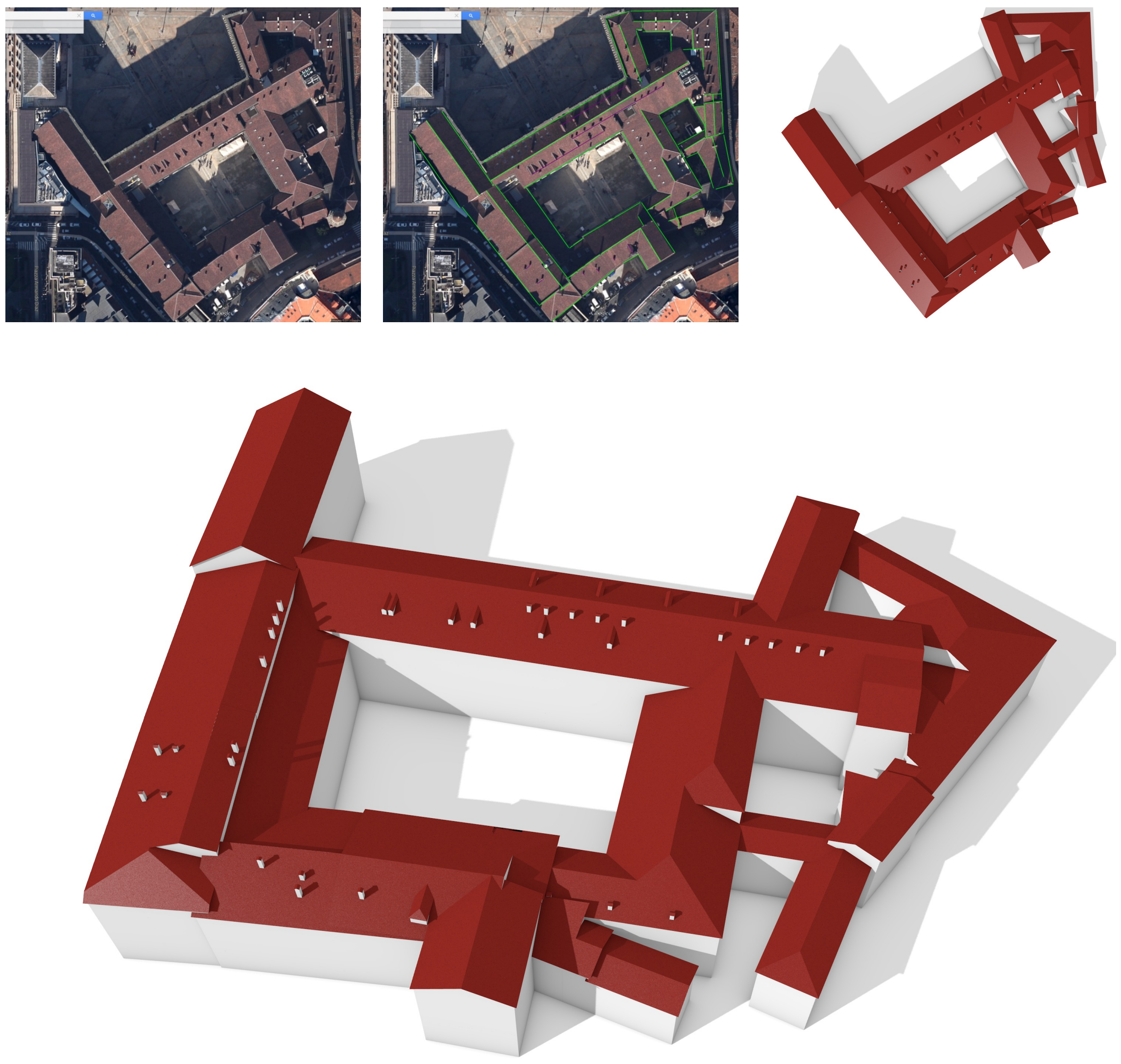


Figure 5.19.: Model of the royal palace of Milan. For this model, the modeling process was done in four steps: 1) obtaining an aerial image of the building (top left); 2) manual tracing of important lines in a vector-drawing program (top middle); 3) extraction of the vector coordinates to the XML format; 4) manual augmentation of remaining parameters in the XML format (like height, roof angle, etc.). The resulting model was created by our prototype implementation, and then rendered (top right and bottom). Aerial images: Imagery ©2015 Google, Map data ©2015 Google.

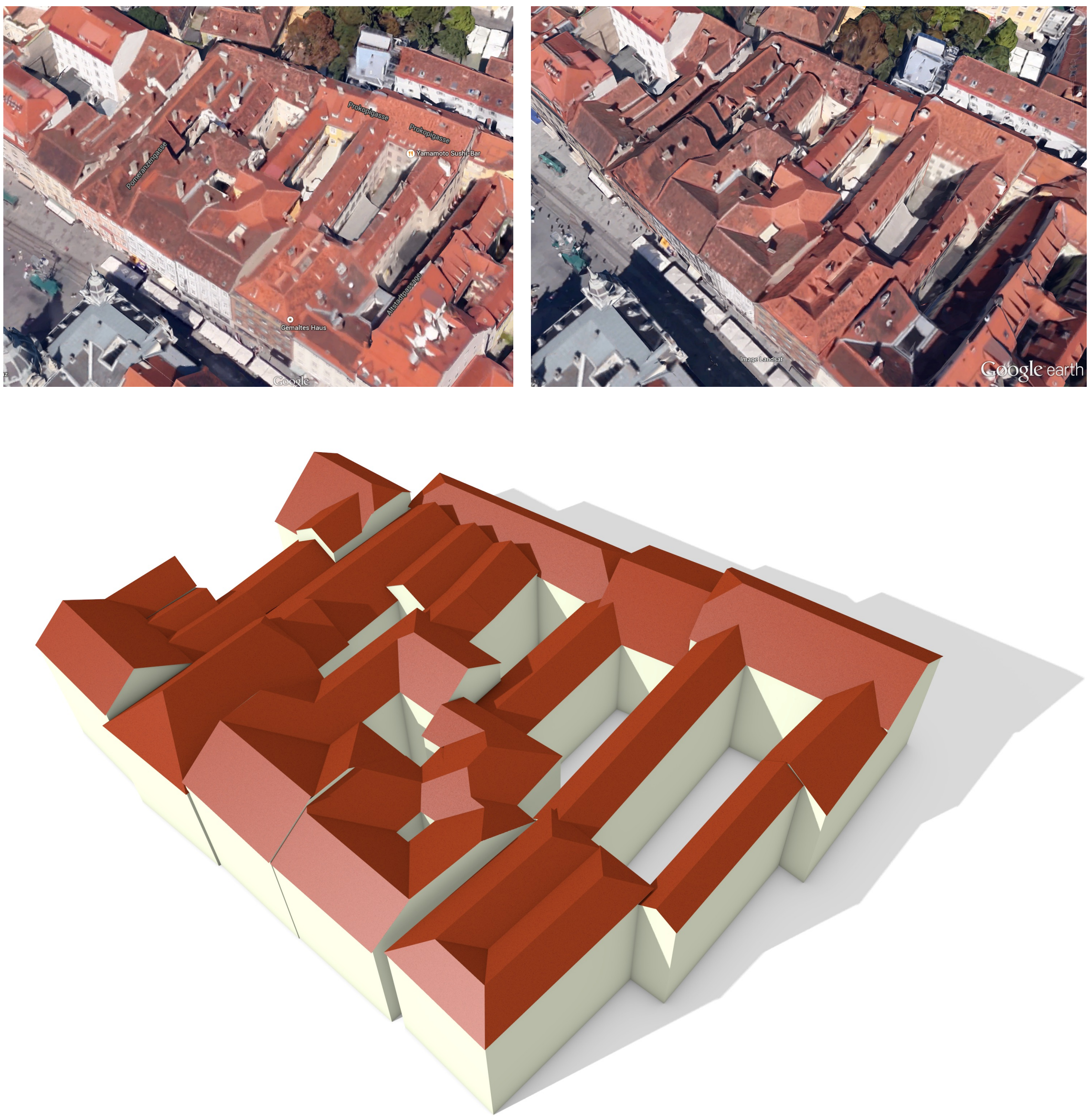


Figure 5.20.: Model of the inner city of Graz, Austria. The detail is the same as in Fig. 5.4. 5.19. Top left: 3D view from Google maps. Top right: 3D view from Google earth. Bottom: Our model. While the models from Google have high detail, they do not ensure constraints like planarity of roof faces or perfectly straight ridges, eaves, or other edges. Our model ensures this by construction. The modeling process was done with aerial images as in Fig 5.19. Aerial images: Imagery ©2015 Google, Map data ©2015 Google.

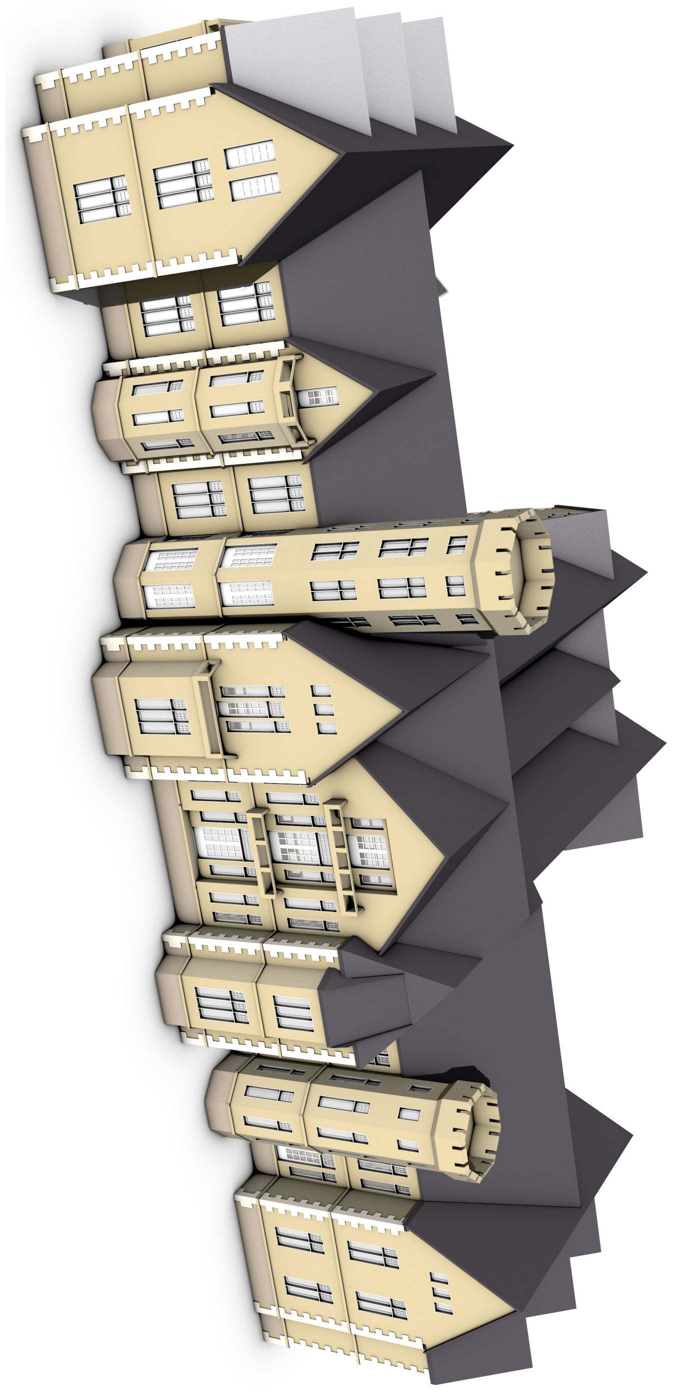


Figure 5.21.: Model of the Magdalena palacio in Santander, Spain (as in Fig. 5.1).

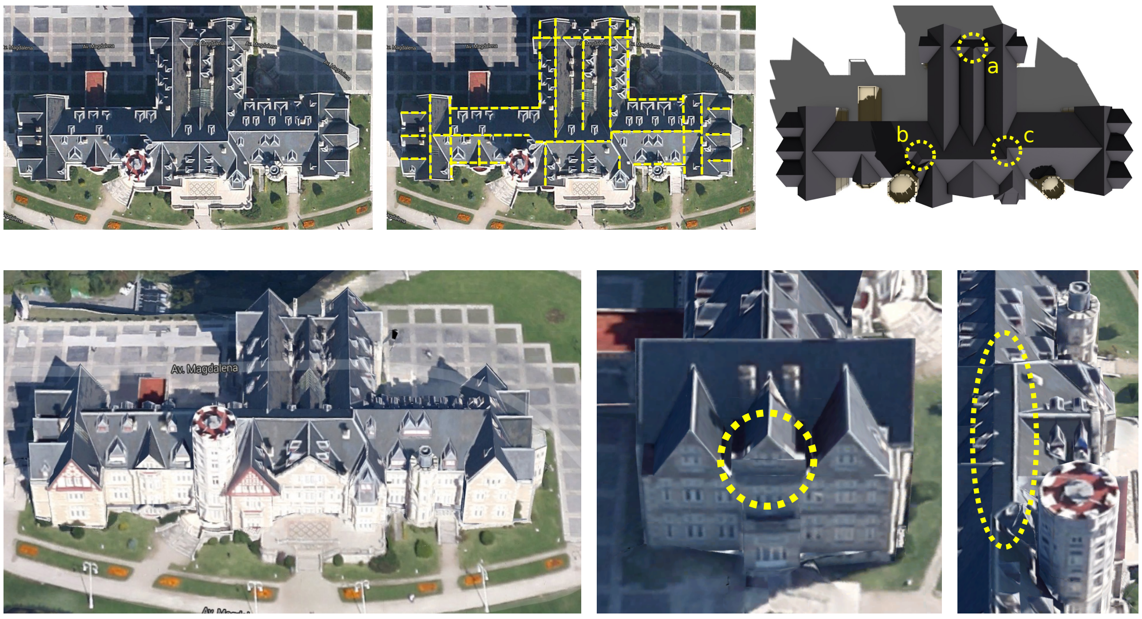


Figure 5.22.: Comparison of our model of the Magdalena palacio (as Fig. 5.21) with the model in Google maps. The modeling process was done with aerial images (middle left, middle middle) as in Fig. 5.19. The detail of the model is increased with existing procedural approaches. Here we used a conventional shape grammar approach for modeling the facades (top). The model has three cases where roof topology changes because of a specified trimming (middle right - marked with red circles). (a) is the top row case from Fig. 5.9, (b) is the middle row case from Fig. 5.9, and (c) is a complex case where five solids with different trimming specifications are involved. The bottom row shows the 3D model in Google maps. Again the constraints of planarity of roof faces and straightness of edges are not satisfied. Two additional regions with wrong geometry are circled in red. Aerial images: Map data ©2015 Google, basado en BCN IGN España.

5.8. Future Work

The main focus in future work is directed towards more automation in the modeling process. One direction is the automatic generation of solid and side descriptions by a rule based system (e.g. stochastic grammars). The other direction is headed towards automatic reconstruction by automatic tracing of important roof lines and the incorporation of additional measurement data for automatic parametrization of building sides (e.g., height of solids). Another interesting direction for future research poses the incorporation of more complex geometry (curves) into the cross-sectional profiles of building sides.

6. Case Study: Procedural Modeling of Architecture with Round Geometry

This chapter was published as the research paper *Procedural Modeling of Architecture with Round Geometry* [26] in *Computers & Graphics*. It was written at the *Institute of Computer Graphics and Knowledge Visualisation (CGV)* [46] at *Graz University of Technology* in collaboration with the *School of Computer Science and Engineering (SCSE)* [76] at *Nanyang Technological University, Singapore*. The paper has been adapted here.

6.1. Introduction

In recent years, manual creation of 3D models has become a bigger and bigger workload for the video game and movie industries. Cultural heritage sites needs more and more 3D models for their visualizations. Procedural modeling provides techniques to automatically generate 3D models, or variants from parameterizable models. Split grammars – a variant that is especially suited for buildings and facades – use a rule system to express repetitions. They usually start with a box surrounding the building and split it into smaller boxes that correspond to various building parts according to the rules. This is highly suitable for facades, because they follow a straight rectangular structure.

Many classical non-rectangular structures also have repetitions. For example, the arrangement of windows, bricks, or pillars in walls, towers, or domes (see Figures 6.1 and 6.2) could be expressed perfectly with the existing splitting idioms, but a standard box-based shape grammar does not allow for these curved shapes or curved arrangements. This places limits on current split grammar

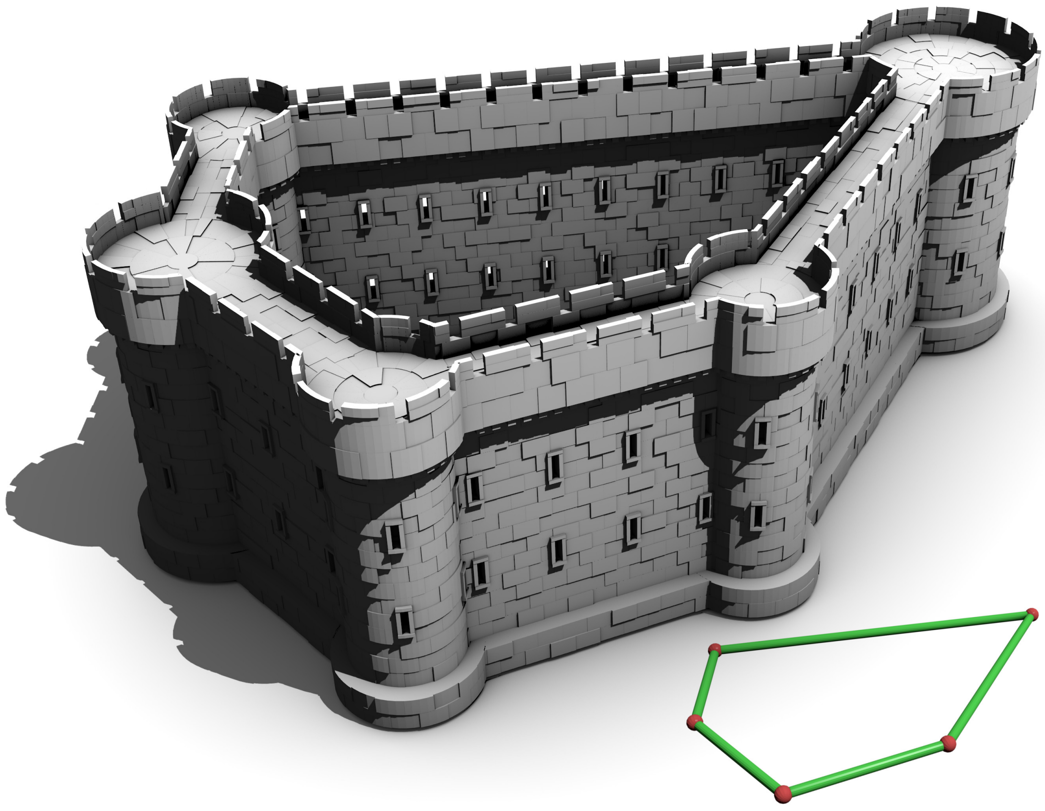


Figure 6.1.: Model of a castle wall, showing procedural splits along straight and round parts. A polyline, used as high-level input to arrange the wall segments and towers, is shown as an inset image on the bottom right. Each node and each edge has additional parameters attached that control the geometry of the final model.

systems and gives the opportunity to introduce new mechanisms to extend their expressiveness.

There are already existing methods that can model or approximate curved and deformed shapes (as in [87, 98]). However, they have problems or fail when it comes to modeling complex or fully curved shapes as in towers, domes, vaults, etc.

We introduce the ability to integrate custom coordinate systems into shape grammars that do not need to be Cartesian. Having, for example, cylindrical or spherical coordinate systems allows us to model such geometry, while keeping the existing splitting idioms and procedural operators that are commonly used to specify procedural models.

The main contributions in this chapter are:

- A method to manually specify and to automatically generate coordinate systems for parts of the procedural model. Common splitting operations then orient themselves along those coordinate systems.
- The design of the procedural system in such a way as to access those coordinate systems and allow for the further modification of geometry (e.g., via Boolean operators).
- The option to feed the system with high level user input (e.g., user-defined line-segments, or graphs), which is used to automatically position geometry. Additional parameters specify the elements and their connections.

Our method is useful for the procedural generation of models of historic or historic-looking buildings, since they traditionally involve many repetitions and a certain formalism with regards to how details are arranged. These properties make them good candidates for procedural modeling, since it can capture the formalism, and also make the models highly parameterizable, while keeping the formalism intact.

In general, objects whose geometry follows that of conic sections and have repeating parts or a certain formalism with regards to the arrangement of parts, are suited to be modeled with our method.

The high-level user input system allows for rapid modeling of connected, repeating structures such as castle walls or gangways.

We will start by giving an overview of the related work, and how existing approaches deal with round geometry. We then show the shortcomings of these

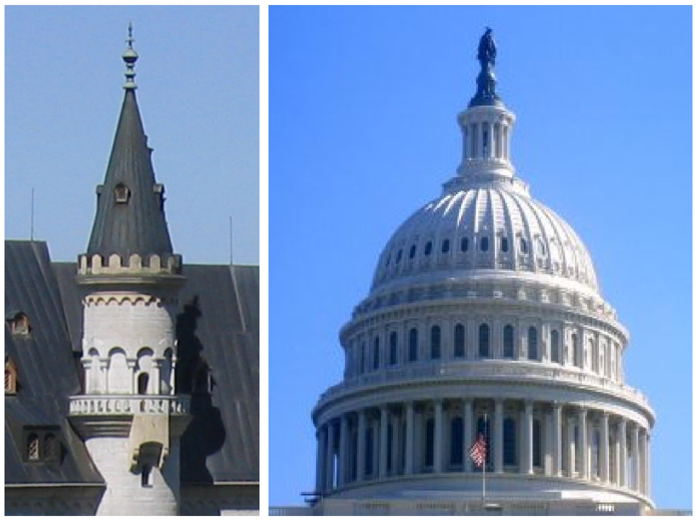


Figure 6.2.: Example of structures that are hard to model with current split grammar approaches. Images are taken from [34] and [35].

methods and present our approach to deal with these problems. An overview of the structure of our system is given, in particular a set of operators is described which allows for effective procedural modeling in the system. To conclude, we will discuss how a model is built, the state of our current prototype implementation, and some limitations of our system.

6.2. Related Work

Shape grammars, a form of procedural modeling, were originally developed by Stiny [78] for the formal definition of paintings. He used sub-shape matching, which identifies shapes and replaces them with other shapes according to rules. The rules are applied iteratively until suitable rules can no longer be found.

In the context of procedural modeling for architecture, Wonka [96] was the first to extend shape grammars to split grammars. In these cases, a bounding scope, usually a box (as in [65]), is generated to contain the final geometry. The bounding scope is then split into smaller parts, according to rules. In this sense, split grammars are a specialization of shape grammars, since the paradigm is to replace a bounding shape with multiple sub-shapes whose disjoint union equals the original bounding shape. The left-hand side of the rule specifies a label for replacement, and the right-hand side specifies a split operation with parameters and labels for new sub-shapes.

Besides boxes, other bounding shapes can be used. For example, [87] uses convex polyhedra, which allows for a much bigger variety of shapes than simple boxes. In [51], free form deformations (FFDs) are used as a new non-terminal class, while [97, 98] integrated FFDs into the grammar so that shapes in the intermediate stages of the grammar execution can be deformed. These extensions offer greater possibilities for expressing non-rectangular shapes, but reach certain limitations, e.g., when fully round geometry such as in towers or domes should be modeled.

The popular split grammar approaches have reached impressive results when it comes to generating mass models. Whole cities with their buildings can be created through application of simple rules. While the focus there lies on the generation of many models, the question of how detailed the models are is neglected to some extent. This leaves room for research into new methods that allow for greater detail, realism, and precision in the generated models. Our approach targets a part of this domain by focusing on round forms, in particular forms that resemble conic sections, which are often used in classical architecture. So far, none of the existing methods have the ability to generate these buildings in very high detail.

The creation of geometry for the connection of different components of a model is a separate concern. In the domain of building roofs, [24] and [25] have shown how procedural roof parts can be connected using high level user input and automatic adaption methods for the created geometry. Our approach describes how geometry is modeled to connect regions of different components using the tools we provide.

There is also some interesting work in fields related to this research. [47] is concerned with different kinds of deformations of objects, which also has relevance with respect to round buildings. [19] and [37] are concerned with segmentation of 3D models and putting segments together in new ways, in order to generate new models or to complete partial models. This can also be useful in the domain of architectural modeling.

We have shown our approach in [27] and extend it here with further explanations, details, examples, and a method to generate models by aligning elements with high level user input.

6.3. Round Geometry in Procedural Modeling and our Approach

6.3.1. Existing Approaches

Round geometry is not very widely covered in the existing literature that is concerned with the procedural modeling of buildings. Split grammars in particular have straight building facades as their original and main application field. That relates to the primitives used in split grammars. Here, we give a brief overview of various methods for split grammars.

Standard box based split grammars (e.g., [96, 65])

The standard split grammar approach uses an axis aligned box primitive. This box is split into smaller sub-boxes and the original box is replaced with the sub-boxes. In this way, the new sub-shapes are also axis-aligned sub-shapes. The splitting happens according to user-specified rules. This process is repeated until no more rules can be applied to boxes. The generated boxes can be used as they are to import geometry, or simply be left empty. That makes up the final geometry of the procedural model.

Boxes are very well suited for the procedural modeling of facades, since facades often naturally have a matrix-like structure where boxes are sufficient as primitives. Simple buildings can easily be modeled by extruding a ground polygon and using a box based split grammar for the facades. In that way, model parts can be created that can have arbitrary detail but must follow a rectangular structure.

Due to the rectangular nature of boxes, round geometry is only covered by the import of pre-modelled asset geometry. This can indeed be any round geometry, but further splitting is then limited. The splitting process only operates along one of the three main axes of the box. Therefore, further procedural refinement of round geometry is very restricted.

Split grammars with convex polyhedra [87]

Here the box primitive is replaced with a convex polyhedron. This brings a much wider expressiveness, since slanted or polygonal shapes can be represented.

It generalizes the box primitive, because convex polyhedra include boxes and the above mentioned splits. A set of convex polyhedra can also represent a non-convex, irregular shape, and round shapes can be approximated by this method.

With this method architectural elements such as arches, winding staircases, or slanted stairs and handrails are possible. Still, the problem remains how a round shape can be further refined. Since there is no reference system (or coordinate system) for the round geometry, splits must be made along a curve, and unnecessary and complicated manual calculations for the setup of the curve have to be made. Additionally, splits are always performed by a plane, which leaves a planar face at the split location, not allowing for round geometry at splits.

Deformation-aware split grammars [97, 98]

These papers concentrate on deformed geometry in split grammars. First, the split grammar for a part is executed on straight geometry in a standard way, and then the result is deformed by a deformation function. This can be done in any level of the shape grammar evaluation, and can therefore generate more than either simple deformed terminal shapes (that cannot be refined further) or a global post-processing of the whole structure.

Structures such as bent walls can be very well described with this method. The splitting of a wall into elements such as windows, doors, or bricks can adapt to the deformed space by automatically varying the number of elements. The splits themselves can be round since they are first performed straight and then deformed.

Although the method is designed to cover round shapes, the approach with FFDs cannot produce exact CSG shapes such as cylindrical or spherical shapes, but only approximations. When a whole cylinder (as in round towers) should be approximated, there is a seam where the ends of the deformed straight parts meet. For a whole sphere (as in domes), there is a seam and a singularity at the top. This is not only a problem for the geometry creation, but splits also cannot create shapes that extend over a seam, since it divides the geometry by construction.

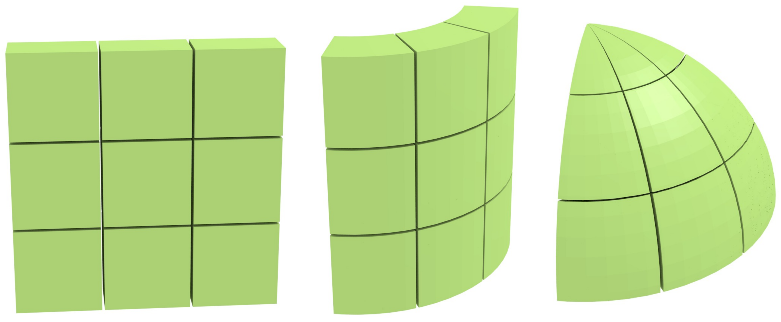


Figure 6.3.: A wall split into nine pieces in different coordinate systems (Cartesian, cylindrical, and spherical). The known splitting idioms from other procedural methods stay the same, only the coordinate systems change.

6.3.2. Our Approach of Custom Coordinate Systems

Our solution for solving these existing shortcomings, is to specify custom coordinate systems in the split grammar by the user. Cartesian coordinate systems produce standard results as in other papers. Cylindrical coordinate systems produce round geometry useful for structures such as towers or pillars. Spherical coordinate systems can be used for domes. Besides that, other coordinate systems are feasible, for example, for cone shaped geometry (i.e. in roofs).

Specifying the coordinate system allows for further splits in the geometry with the familiar splitting idioms. Where a split of a wall in a common split grammar produces rectangular parts, we can now also split cylindrical or spherical walls into sub-parts (see Figure 6.3). Because we do not loose the reference system, the refinement of round geometry has no limits. This is an advantage of our method in contrast to the common pre-modeled asset import, or the system with convex polyhedra.

In the edges where round parts with different coordinate systems meet (e.g., in round cross vaults), the coordinate systems of both parts are reference-able. This is needed, since the geometry in the edge is related to both coordinate systems.

6.4. Common Procedural Modeling Systems and Our Proposed System

6.4.1. Existing Approaches

For the procedural generation of architecture there are a few common methods which can be found in scientific literature:

Split grammars

Split grammars are a popular method to generate buildings, especially the facades [65]. They use simple rules given by the user to define the partitioning of geometry. Split rules divide the geometry into multiple parts along one axis. This works very well for rectangular elements that are arranged in an array or matrix-like fashion, such as in the case of facades. When the structure gets more complex, additional operators have to be included into the rule system (see [77]).

The problem is that when the modeling becomes more complex, more operators need to be added to the split grammar. This leads to the extension of the grammar with common programming operators, known from general purpose programming languages. Thus, the split grammar becomes more and more like a general purpose language. One can argue, that this is a case of ”re-inventing the wheel”. Further rule specification by the user becomes more and more like programming, as in a general purpose language.

Scripting

The scripting approach simply uses a general purpose programming language and augments it with features and operators for procedural modeling. An example is Leblanc’s paper [53]. The advantage is, that all features of the general purpose language are available, and no reimplementation of them has to be done.

The disadvantage is of course, that in simple cases (as for many facades) the simple rule system is not available for the user. However, the code for modeling a building often follows certain patterns, and these patterns may very well be encoded in a rule system. This rules system could then be like those mentioned in Section 6.4.1 and reproduce a part of the scripting language. A parser could now transform the rules into the scripting language (such as in [51]).

For this approach, the choice of the language is important, since its features and idioms influence how easy modeling becomes for the user. Generally, modern scripting languages are a good fit for this task.

Others

Other approaches to modeling buildings also exist. [48] use a modified straight skeleton algorithm to generate the outer building shell. He uses a variant where he can adjust the 3D slant of each face in arbitrary stages, which allows him to generate complex wall and roof geometry. This method can produce many complex building forms, but has no system to describe repetitions built within. Modeling a lot of detailed elements (such as bricks) is difficult, and in general, straight skeleton algorithms are prone to numerical errors, which makes their implementation difficult.

Finkenzeller [31] also uses polygonal floor plans to describe a building. In contrast to [48], he does not use the straight skeleton, but only extrudes the walls straight from a polygonal floor plan. However, he uses multiple floor plans per building and can link them together, allowing for complex forms for buildings. There is also no handling of repetition of detail.

6.4.2. Our Procedural Modeling of Round Geometry

The purpose of setting up coordinate systems for the splitting process is to keep the existing splitting idioms unchanged, and to modify only the space in which they are operating. This lets the user work with well known methods and quickly create procedural models with round geometry. However, it gets more complicated as the complexity of the geometry increases. Especially in the areas where round and straight geometry meet, or more generally, where parts with different coordinate systems meet. In such cases, more advanced operations have to be carried out. This usually includes Boolean operations in order to trim the geometry. To satisfy these complex needs, we chose to implement our system with the very flexible scripting approach, augmented with features and operators similar to that of split grammars.

Components in the System

In order to describe the structure of our scripting system, we give an overview of its main components here:

- **Coordinate System.** The coordinate system is the main component where geometry orients itself. We let the user set up a coordinate system by specifying its type (Cartesian, cylindrical, spherical, or basically any 3-dimensional coordinate system) and the needed parameters (origin, vectors for alignment of axes, etc.). Each one of the three coordinates has a name (X, Y, Z, ρ, ϕ, etc.). The splits that will be performed then usually split via a coordinate surface for a coordinate. A coordinate surface is the surface formed by the set of all points in the coordinate system, where one coordinate is fixed and the other coordinates can vary arbitrarily. In the case of a Cartesian coordinate system, the coordinate surfaces are all axis-orthogonal planes, producing the standard splits known from box-based split grammars.

- **Scope.** The scope describes the "abstract" form of the shape. It is the basis for further split operations - the splits themselves create new sub-shapes with modified scopes. Usually, the scope is a box, but can also be a convex polyhedron as in [87]. In our system, it is usually a box in a specific coordinate system - meaning for a Cartesian coordinate system, a normal box, but for a cylindrical coordinate system, a bent box (bound by: inner and outer radius, bottom and top plane, and two side planes with an opening angle - also see Figure 6.3). Additionally we allow scopes to be bounded by less or more than three dimensions, infinite scopes, or scopes that are bounded by different coordinate systems, resembling (possibly bent) convex polyhedra.

- **Shape.** In accordance to [65] and [77], we define a shape as a procedural entity, having a scope and various attributes. The final geometry of the shape can be the geometry of its scope, a modification of the scope's geometry, imported pre-modelled assets (that orient their dimensions along the scope), or simply empty geometry.

- **Split Tree.** The split tree results from the application of operators to the shapes, because they produce new sub-shapes that are inserted into the split tree as child-nodes. However, the split tree can also be created or altered manually.

- **Operator.** Our system has various operators that can modify the different components. Most important are the shape operators, which are similar to the common operators in split grammars (first and foremost, the common split operator). A shape operator generates new sub-shapes (child-nodes in the split tree) with specific dimensions in the specified coordinate system. Then, it copies all attributes from the current shape to the sub-shapes, and calls the specified rule functions for each sub-shape. Other operators modify the scope or coordinate system of a shape. For a more detailed description of operators see Section 6.4.2.

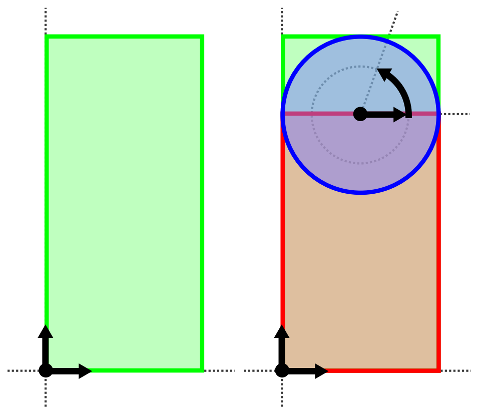
- **Rule Function.** Rule functions are normal functions in the programming language that are called by operators. The operator creates a new shape, and the rule function then works locally on that shape. This means, for example, that any attributes that are directly accessed come from this shape, and any operators that are called in the rule function work on the scope of this shape. Through this mechanism, the rule functions can be implemented as ordinary functions in the general purpose language. Because of that, the user can take advantage of all existing programming features, while still using common splitting idioms known from split grammars.

- **Alignment element.** High-level alignment elements provide a way to align procedural elements. This can be points, lines, polylines, graphs, etc. Additionally, parameters can be given for every part of the alignment element (e.g. for every point or line of a graph), so that the aligning procedural element can be adapted to specific needs.

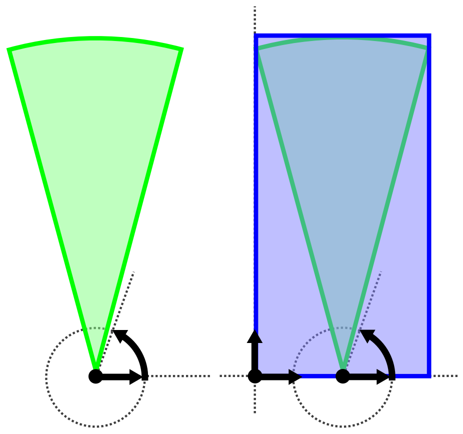
Operators

We have a multitude of operators in our system that allow for the modification of shapes, scopes, and coordinate systems. Here, we present a list of some of the most important operators along with a short description.

- **subdivide**. Known from common split grammars. It divides a shape according to its scope along one coordinate of its coordinate system into a fixed number of sub-shapes. For each sub-shape, the dimension of its scope along that coordinate is given by the user. Possible are absolute values, as well as relative values (they specify, for example, that sub-shape *a* has double the



(a) Cartesian coordinate system (left), and additional generated cylindrical coordinate system (right).



(b) Cylindrical coordinate system (left), and additional generated Cartesian coordinate system (right).

Figure 6.4.: Operators are used to generate new coordinate systems and scopes. For (a) wrap-cartesian-over-cylindrical, and for (b) fit-cartesian-into-cylindrical is used. Black arrows symbolize coordinate systems, blue and red boxes symbolize scopes of new shapes.

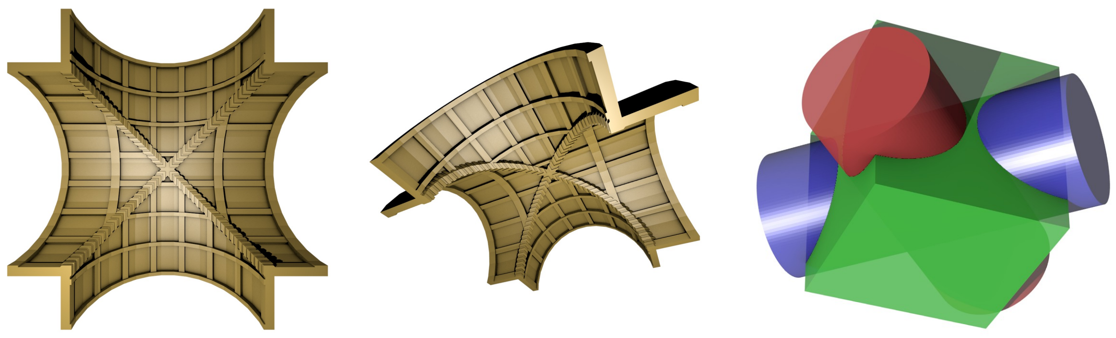
dimension along that coordinate as sub-shape *b*).

- **repeat.** Also known from common split grammars, and similar to subdivide. The difference is, that a shape is split into a repeated set of sub-shapes, which have a fixed dimension along one coordinate. The sub-shapes are repeated such that all the space of the original shapes scope is used.

- **call-rule.** Simply generates a new sub-shape for a shape with the same scope (and no splitting), and then calls the specified rule function. This is useful when rule functions are reused, and called from multiple locations in the code.

- **bounds-expand.** Expands the dimensions of the scope by given values for the three coordinates.

- **fit-cylindrical-into-cartesian.** This operator takes a shape with a Cartesian coordinate system and creates a new sub-shape with a cylindrical coordinate system that is fitted in at the top position (see Figure 6.4a). This is useful



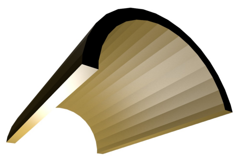
(a) Cross vault bottom view. (b) Cross vault side view. (c) Coordinate systems for the cross vault. Two cylindrical ones (red and blue), and a cartesian one (green).

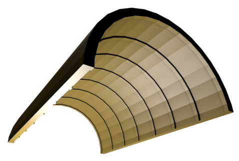
Figure 6.5.: Cross vault, showing repetitive geometry at a round surface. Two barrel vaults (that are rounded differently) cross each other. The decoration on the edge where both vaults intersect is modeled using the coordinate systems of both barrels. The two vaults use cylindrical coordinate systems, and for the cutout and the decoration an additional Cartesian coordinate system has to be set up (Figure 6.5c).

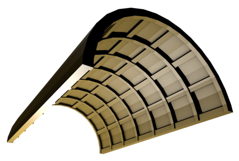
for building arches, since the top part of the arch can now be modeled with shapes following the cylindrical coordinate system. For the lower part of the arch, an additional sub-shape with the original Cartesian coordinate system, but with a shorter scope is generated.

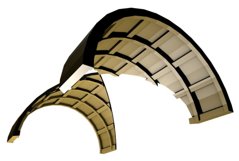
- **fit-pointed-arch-into-cartesian.** This operator is similar to fit-cylindrical-into-cartesian, but generates two sub-shapes with cylindrical coordinate systems, whose origins are displaced to the left and to the right. This lets a user model pointed arches, such as those in Figures 6.9 and 6.7.

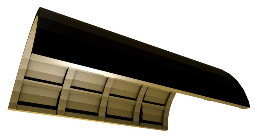
- **wrap-cartesian-over-cylindrical.** Here, for a shape with a cylindrical coordinate system, a new sub-shape with a Cartesian coordinate system is generated. The sub-shape's scope is dimensioned in such a way, that the original shape's scope is circumscribed (see Figure 6.4b). This is useful when rectangular elements have to be inserted into round objects, such as windows in cylindrical towers.

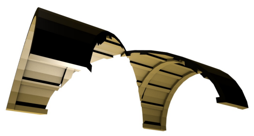

(a) The first vault is created for one direction (lines 3, 20).

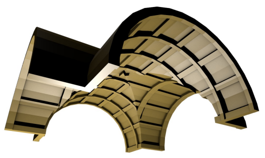

(b) The vault is split along the axis (line 29).


(c) The bigger parts are further split and form the cassettes (line 45).


(d) Geometry is cut out in order to make room for the second vault (line 34).


(e) The second vault is created and split analog to the first one (lines 11, 21).


(f) Similarly, geometry is cut out of the second vault.


(g) The two vaults are joined (line 23 - implicit).

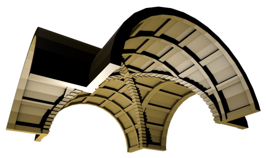

(h) The decoration is added (lines 22, 23).

Figure 6.6.: Steps in the generation of the model in Figure 6.5. One barrel vault is generated with a specified coordinate system, details are added via split rules, and geometry is cut out (top row). The same is done for the other barrel, both vaults are joined, and the decoration on the edges is added (bottom row). The line numbers refer to lines in Listing B.1 that are crucial for this step of the generation.

6.5. Workflow

The workflow of our system is similar to that of working with a standard split grammar. We describe the workflow via the example of a cross vault (Figure 6.5) that we have modeled with the system. Listing B.1 shows a pseudo-code for the cross vault, and Figure 6.6 shows various steps in the generation of the model. Table B.1 lists all the attributes that are present for one shape in the implementation of our system, and that are accessible in the code.

Before starting, we determine the main parts of the model. In this case, these are the two barrel vaults that intersect with each other. For each of them a cylindrical coordinate system is generated, and an additional Cartesian one is needed for trimming and for the decoration along the edges.

The pseudocode in Listing B.1 shows how the rules for this model look. We have programmed our system with our own in-house language (GML [38]), so we show only pseudo-code (in a python-like syntax). The first rule *Start* sets up the coordinate systems (one cylindrical for every barrel vault, and one Cartesian for the decoration and trimming geometry). Then it calls the two rules for each vault direction (*Vault0*, *Vault1*) and the decoration (*Decoration*). The rule *Vault0* generates one vault with normal split rules (Figures 6.6a - 6.6c), and then cuts out the part where the other vault will join (Figure 6.6d). For the other vault the same is done (Figures 6.6e, 6.6f) and both are joined (Figure 6.6g). At last, the decoration in the rule *Decoration* is created. Here we use for the first time coordinate-bounds from different coordinate systems for the scope of an element (Listing B.1 - line 69). Then at the end, all elements are joined to form the final model (Figure 6.6h). In the code (Listing B.1 - line 23) this happens implicitly, since the geometry of all the sub-shapes in the *children* attribute is automatically joined to build the final geometry.

6.6. Further Methods

6.6.1. Automatic Generation of Coordinate Systems

One of the central elements of our method is to create shapes with different coordinate systems automatically. We can automatically place new shapes with new coordinate systems into an existing one. Figure 6.4 shows two cases.

Figure 6.4a shows an operation that is useful for the generation of arches. Here the operator fit-cylindrical-into-cartesian takes a shape (green) with a Cartesian coordinate system and generates two new shapes. The first one (red) is with the same coordinate system, but with a modified scope that only extends up to the point where the arch starts. And the second one (blue) has a new cylindrical coordinate system which fits in and is centered at the top of the old shape. This shape can then be used to model the actual arch. For example, radial splitting can be used to model single bricks in the arch.

Figure 6.4b shows an operation that is useful for aligning elements in a circular way. A shape with a cylindrical coordinate system can be used and split radially. The operator wrap-cartesian-over-cylindrical creates a new shape with a new Cartesian coordinate system. The scope has dimensions such that the new shape is wrapped around the original wedge-like shape. If this is done for many of the wedge-like shapes, the new shapes will all have Cartesian coordinate systems that are aligned in a circular fashion. They can then be further split to generate detail, or new coordinate systems can be fit in, in order to generate elements such as circular arranged pillars or arches.

6.6.2. Customizable and Reusable Modules

The high degree of customization through parameter arguments or modification of code parts also makes models very reusable.

In Figure 6.7, the model of a cross vault is varied by two high-level parameters. Many of these cross vaults can now, for example, be combined to form gangways (Figure 6.9). Here, only the parameters of the same axis of adjacent cross vaults have to match, the other axis can differ arbitrarily.

6.6.3. High-level Geometry Specifications

When multiple modules are combined, high level alignment elements can be used to align and adapt them. Additionally, parameters can be specified that allow for the adaption of the individual modules. This provides a way to create and modify the model on a significantly higher and easier level. That means that even inexperienced users can modify the model and create different variations on a coarse but not detailed level.

We have implemented the processing of polylines and graphs as high level input. Examples show a castle wall that is automatically placed along a polyline (see Figure 6.1) and the roofing of a gangway with a section where multiple gangways come together (arranged by a graph structure - see Figure 6.10). The polyline and the graph structure that control the high level geometrical alignment are also shown in the Figures. Each node and each edge of the polyline or graph have additional parameters attached that control the appearance of the model there.

This means, that the creation of such models can be split into two parts. One, where experienced users create a model with a high-level primitive and attached parameters. And one, where inexperienced users can simply modify the high-level primitive and the parameters. The second part is significantly easier and is possible for new users who are not familiar with the system.

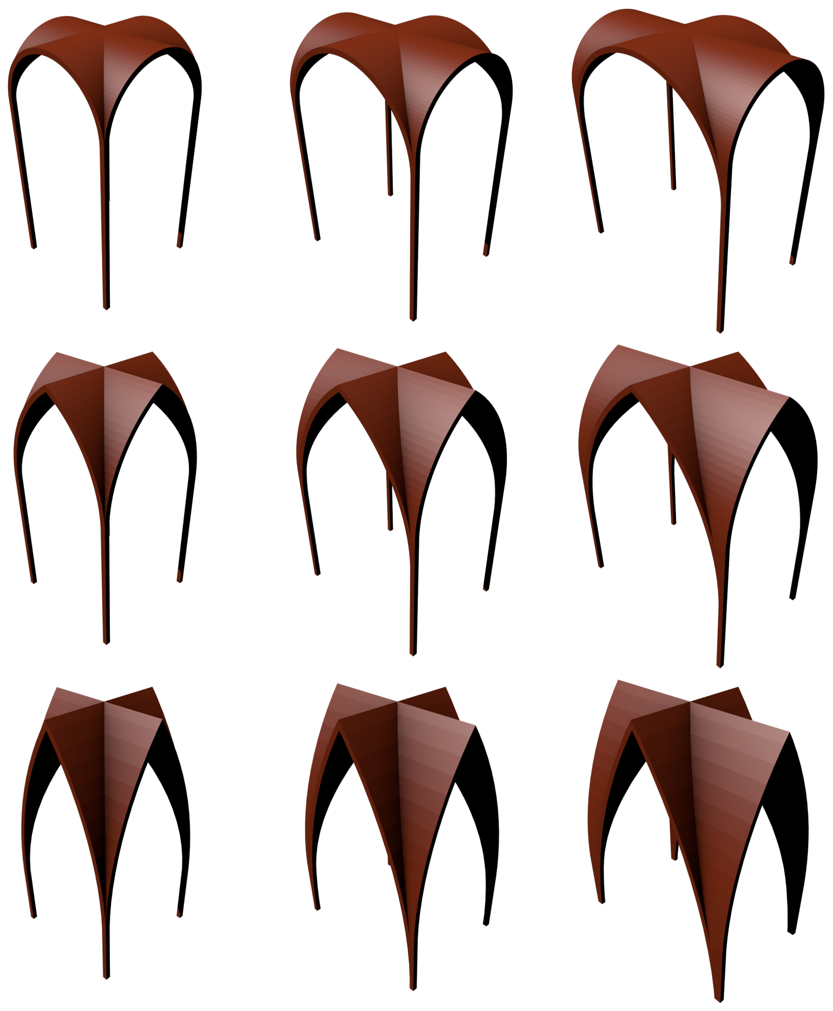


Figure 6.7.: Variations of a procedural cross-vault model. To the right the width increases, and the arch automatically adopts to the new space. To the bottom the angle of the arch decreases and the arch becomes a pointed arch.

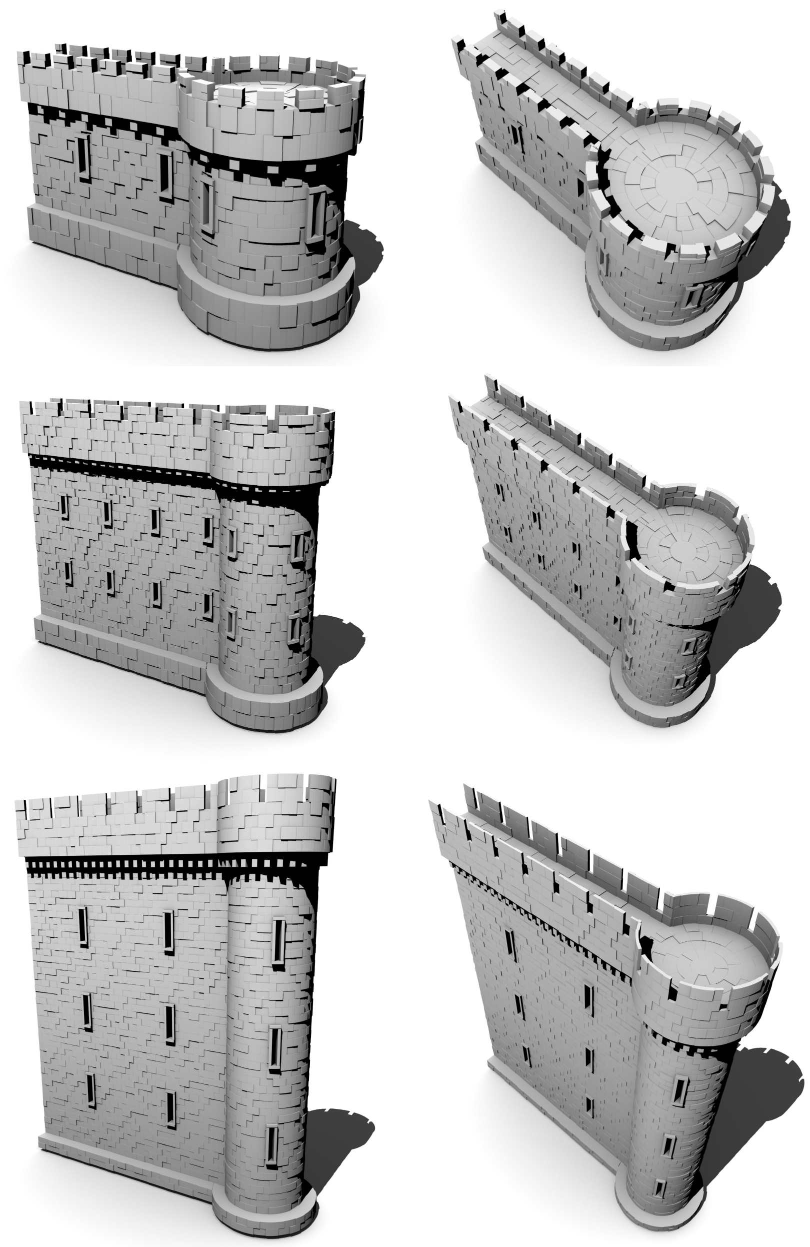


Figure 6.8.: Model of a segment of a castle wall. Each row shows the model with different parameters. Splits that subdivide an area can be performed on rectangular geometry (straight wall part), as well as on round geometry (round tower part) in the same fashion. The bricks are actual 3D geometry and have been modeled by using repeat splits.

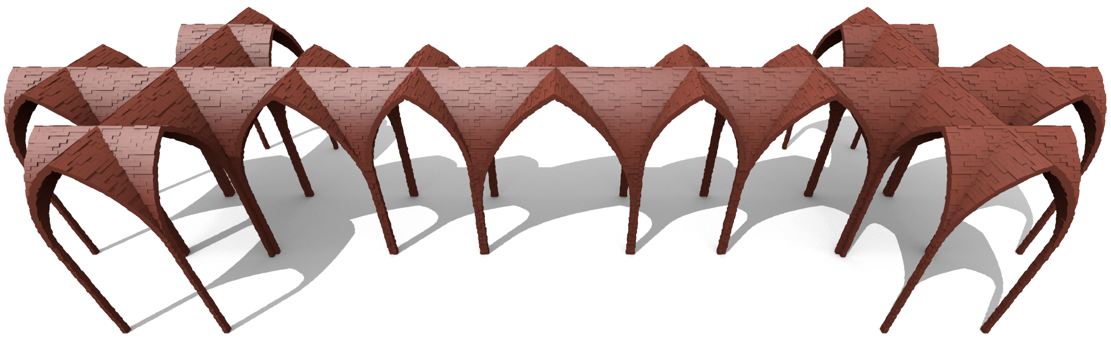


Figure 6.9.: The cross vault from Figure 6.7 can be easily repeated to form gangways. The length and width of a procedural cross vault are independent, which allows them to be combined in interesting ways. Additionally, the bricks have been modeled with repeat splits.

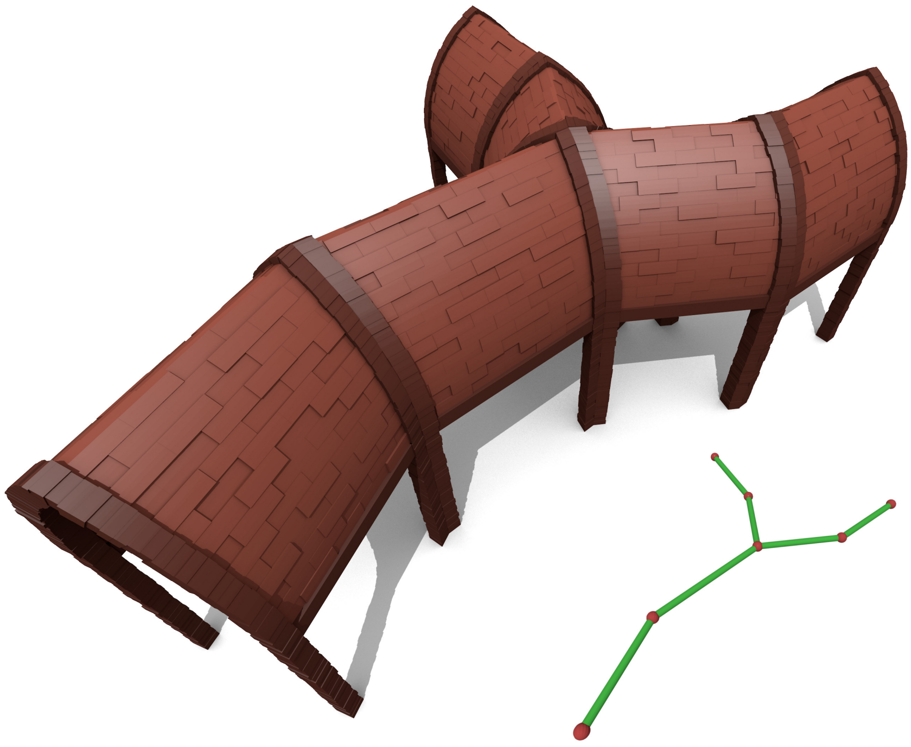


Figure 6.10.: Model of a roofing for a gangway. Elements are aligned according to a graph, with a multi-way crossing in the middle. A vault is placed over every edge of the graph and the pillars are created for every node. The graph is shown as an inset image on the bottom right.

6.7. Implementation

A challenge that our approach experiences in comparison to normal shape grammars is that we must be able to address specific coordinates and coordinate systems and switch between them. For example, the cutouts and the decoration in Figure 6.6 are modeled by referencing different coordinate systems. Also, boolean operations are often needed to cut out geometry, for example, the windows in Figure 6.11. In an ordinary split grammar with rectangular geometry, the hole for a window can often be modeled by simply splitting a wall first horizontally and then vertically, leaving the resulting inner element simply empty. With round windows this is no longer possible, and round geometry has to be created that is then subtracted via a boolean operation from the wall geometry. Due to these factors, a simple grammar is no longer sufficient and we take a more general approach to programming our models, which is still inspired by split grammars in order to ease modeling.

We implemented our prototype in GML [38], our in-house procedural modeling language. As described in Section 6.4.2, we have rule functions, which are implemented as ordinary functions in GML. The splitting operators call the functions and set the programming scope to the newly generated shapes. Inside a function the user can program with all the available features of the language.

Every shape has a set of attributes. When called, each rule function creates a new shape and the attributes of the parent shape are copied to the newly created one. Table B.1 shows a table of the attributes of a shape. Every attribute can be modified by the user. Listing B.1 shows pseudo code for the model in Figure 6.5.

In general the user first creates one or more coordinate systems in the start shape by setting the *coordsystem* attribute. He then defines the scope of the shape by setting the dimensions for each coordinate in the *bounds* attribute, and setting the *scope* attribute to the names of the coordinates that are used to bound the shape. He can save arbitrary data in the *user* attribute, and access it in subsequent rule functions. He can now split the shape with splitting-operators that save the resulting sub-shapes in the *children* attribute. The *parent* attribute is automatically set to the (parent) shape whose rule function created this shape. Finally, by default the *geometry* attribute is automatically set to the union of geometries of all children. This, however, can be overridden by the user. He can manually take the geometry from various children and perform Boolean operations on them, and then save the result there. Alternatively, he can even save some arbitrary geometry there. After the rule function is executed, the geometry from the *geometry* attribute is passed up to the parent shape.

As with other split grammars, our method splits a shape into multiple sub-shapes and thereby generates new scopes for them. In our implementation there is no geometry involved so far, everything simply operates on the parameters of the shapes/scopes. Later, leaf shapes are filled with geometry. This process is fast, since geometry creation is a local process here. After that the parent-shapes can access this geometry. By default, parent shapes only collect the geometry of their children and pass it along the hierarchy to the top. However, they can also implement their own behavior, for example perform boolean operations on the geometry that they receive from their children.

The geometric kernel that we use is based upon convex polyhedra, where solids are represented as a collection of convex polyhedra. It is also our in-house development, and it allows for fast and exact boolean operations on solids. Exact also means, that we do not get topological problems when we do multiple, complex boolean operations. We approximate round shapes with a reasonable amount of convex polyhedra, and can set parameters for the level of detail.

The final models were then exported as .obj files and the renderings were done in blender [18].

6.8. Limitations

6.8.1. Solid and Mesh Limitations

The type of geometric data structure that is used determines the operations that can later be performed. While a data structure representing solids makes it easy to perform boolean operations (union, intersection, difference), shapes that are more free-form shapes which do not follow the geometry of conic sections can be difficult or impossible to reproduce, and may need elaborate approximation. On the other hand, a mesh data structure is well-suited for the representation of arbitrary forms (or for their approximation to a certain degree), but is often error prone when it comes to boolean operations.

Since we need boolean operations in the more complex structures, we decided on a solid modeling data structure. This limits the forms generated. In fact we do not have any examples of spherical models, since our approximation methods for them are currently still slow. However, with the right data structures and algorithms, the set of primitive forms that are possible, can be extended.

6.8.2. Performance

The possibility to perform Boolean operations on the created geometry allows for the creation of a wide variety of shapes, but can also result in serious costs in terms of performance. Because the system only creates geometry in the leaf nodes of the derivation (as described in Section 6.7), it is a fast and local process, but Boolean operations can work with arbitrary complex geometry, and therefore become a time-consuming process.

For example the bricks in the wall in Figure 6.1 are all modeled with split rules and create a lot of geometry. Cutting out the window openings from the fully generated brick wall now becomes the most time consuming part of the evaluation of the model, because there is so much geometry involved.

However, this depends on the method with which Boolean operations are implemented in the system. Using screen-space CSG, for example, would probably significantly reduce the cost of Boolean operations on geometry-rich parts.

6.8.3. Modeling Edge Decorations

We have shown that it is possible to model edge decorations because we can access all the different coordinate systems of multiple intersecting elements. For example, in Figure 6.5 we modeled a cross vault and decorations along the intersecting edges of the vaults. For the decorations access to the coordinate systems was needed, but this was not enough. Additionally, for each decoration along an edge, we needed a Cartesian coordinate system in order to bound the elements of the decoration (see green coordinate system in Figure 6.5c). This works, because both intersecting vaults have the same dimensions, and intersect along two edges that each lie in a plane. If the vaults had different dimensions, the intersection would be more complicated. In this case, a normal coordinate system is not sufficient to model the details, and some kind of moving coordinate frame would be needed.

6.9. Results and Applications

6.9.1. Castle Wall

For the model in Figure 6.8, a Cartesian and a cylindrical coordinate system are set up for the respective parts. Additionally, various parameters that determine the dimension of different parts are specified. Then, rule functions for both main parts are called separately, where the splitting process begins. Splits are specified by the user for the circular tower part simply like for the straight wall part. In fact, in this model many rule functions for details (windows, bricks, crenelations, etc.) are used in both branches (wall and tower), and do not have to be specified twice. Where needed, differences for both branches can be resolved with conditional statements in the rule functions. This shows that rule functions can be reused, and that the splitting method is independent of the coordinate system. The bricks, windows, and pinnacles have all been modeled with repeat rules, such that the number of elements automatically adapts to the space given.

The area where the wall and the tower join, needs special handling. Because we have set up the two coordinate systems in advance, we can reference them from both branches. This allows us to construct volumes that are used to cut out holes in each part (via Boolean operators), so that then, both parts fit perfectly together (producing a disjoint union as result). From the straight part segments are cut out that correspond to the cylindrical coordinate system, and at the round part at the top the pinnacles are trimmed with geometry corresponding to the Cartesian coordinate system.

The full castle wall (as in Figure 6.1) uses the high-level system described in 6.6.3. The user can place a polyline on the ground that is used to align the straight and round wall parts. Straight walls are created along the line segments, and round towers are created in the corners.

6.9.2. Neuschwanstein Castle Towers

Figure 6.11 shows photos of two towers which are part of Neuschwanstein castle in Germany. We have created models which are inspired by them. They illustrate the technique described in Section 6.6.1. Additionally, a visualization of most of the created coordinate systems in the two models is shown.

For the tower on the left, first the main cylindrical coordinate system is set up. It is split into a bottom, middle, top, and roof part. The parts are refined with further splits. Then there are some parts that need additional coordinate systems. These are, for example, all the arcs that exist in the model. For these, the process is always the same. First, the main cylindrical coordinate system is split radially into multiple wedge-like shapes. Each of them will later contain one arc. Then, for each wedge, a shape with a Cartesian coordinate system is wrapped around (as in Figure 6.4b). The shape is now used to fit a shape with a cylindrical coordinate system vertically (as in Figure 6.4a). This last shape is then further used and refined to model the arc, or to model geometry that represents the inner, empty part of the arc which is later used for Boolean subtraction.

For the tower on the right, the process is similar but it involves two separate towers that are joined together.

6.9.3. Gangway

The gangway model (Figure 6.10) uses a graph as high-level input to align its elements. For each line segment a vault is created. At the nodes, additional Cartesian coordinate systems are created (such as the green Cartesian system in Figure 6.5c), that are needed for the edge geometry and the pillars. These coordinate systems are also needed to trim the vault geometry at the nodes. Their alignment is along the angle bisectors of the line segments that meet in one node. The pillars are modeled in a similar way as the decoration along the edges in Figure 6.5

6.10. Conclusion

We have described and illustrated a method that allows for the definition and further refinement of round geometry in procedural modeling. This approach

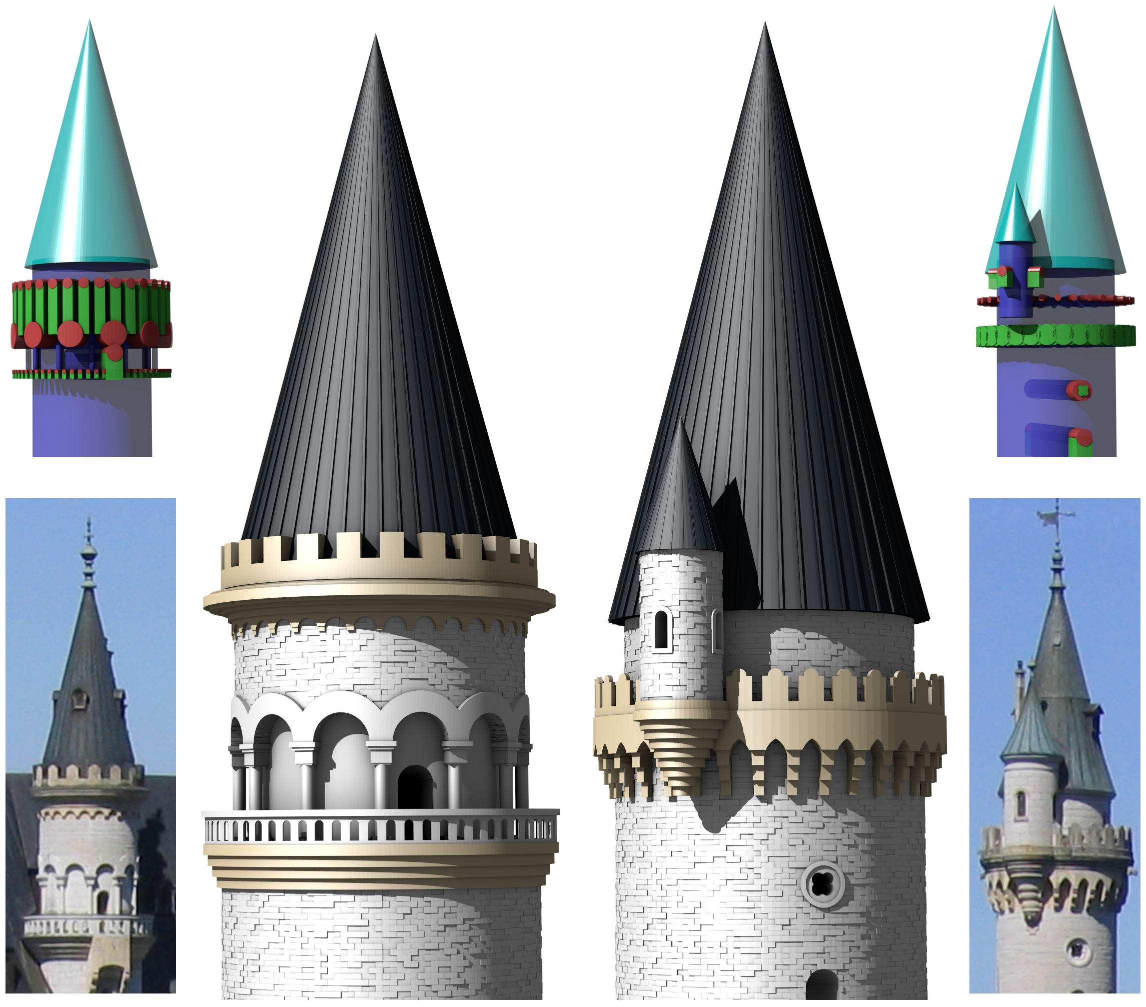


Figure 6.11.: Towers of Neuschwanstein castle in Germany. The photos (sides - bottom, taken from [34]) show two different examples of the castle's towers. The respective 3D models (center) have been created with our method. Not all elements have been modeled, nevertheless the highly detailed results of our approach are visible (e.g., the bricks and all bumps are actual 3D geometry). The models have complex setups with many nested coordinate systems that are generated via the method (sides - top). Cartesian coordinate systems are indicated via boxes, cylindrical ones via cylinders, and conical ones via cones. Not all of them are shown, since this would overload the depiction. The technique in Figure 6.4 is used multiple times to generate the coordinate systems.

is new and has not been explored in this form so far. Existing methods are limited when it comes to detailed generation of procedural geometry which needs further refinement. Our approach allows for the procedural modeling of geometry following conic sections, which are often present in historical buildings. We have shown that our approach can adequately represent at least parts of their geometry.

Furthermore, we have described a method to let users specify high-level input in order to arrange automatically generated procedural elements.

6.11. Future Work

Our method provides a basis to specify round geometry for procedural models. The following subsections give example areas where this concept could be extended.

6.11.1. Profiles and Extrusions

Giving the user the possibility to define profiles in the system would allow for solid building primitives that are generated by extrusion or revolution. They fit perfectly into Cartesian, respectively cylindrical coordinate systems and could be fully integrated into the splitting process. This would enhance the form expressiveness of the system.

6.11.2. Parametric Surfaces

Our approach could naturally be extended to parametric surfaces. A parametric surface already has two coordinates (input parameters), and the third can be augmented. This would most likely be the distance from the surface (along the surface normal at that point). Using this method, it would be possible to model modern buildings that have a building shell in the form of free-form surfaces. These free-form surfaces could be parameterized and then further refined with our method.

6.11.3. Ornaments

A possible method to model ornaments in architecture, is to model an ornamental pattern in a 3D software such as Blender and save the geometry. This geometry is then inserted into the terminals of a split grammar and transformed accordingly. Because the terminals in our system cannot only be boxes, but also round shapes, we can extend the ornamental pattern to round, circular, spherical, or other geometry.

6.11.4. Mechanical parts

Many mechanical parts have geometry following conic sections and consist of details and sub-parts that are placed according to exactly specified dimensions. This specification formalism can also be described with procedural modeling, and our modeling method should be possible to extend to cover a part of this domain.

7. Case Study: Curvature-controlled Curve Editing using Piecewise Clothoid Curves

This chapter was published as the research paper *Curvature-controlled curve editing using piecewise clothoid curves* [41] in *Computers & Graphics*. It was written at the *Institute of Computer Graphics and Knowledge Visualisation (CGV)* [46] at *Graz University of Technology*. The paper has been adapted here.

7.1. Introduction

The creation of good-looking curves is a fundamental task in 2D (and 3D) shape design. Through a recent collaboration with the surfacing department of a car manufacturer we realized how difficult it is in practice to convert a given design curve into a high-quality parametric curve that is suitable for further processing in a high-end CAD system (CATIA). Observing the work of the surface engineers we developed three hypotheses about their requirements:

Hypothesis 1 *The control polygon alone must unambiguously, and in a predictable way, define the shape.*

Hypothesis 2 *The control polygon must be as sparse as possible (a) to offer control and (b) to avoid artifacts.*

Hypothesis 3 *Controlling curvature is essential; much time is spent on removing curvature artifacts.*

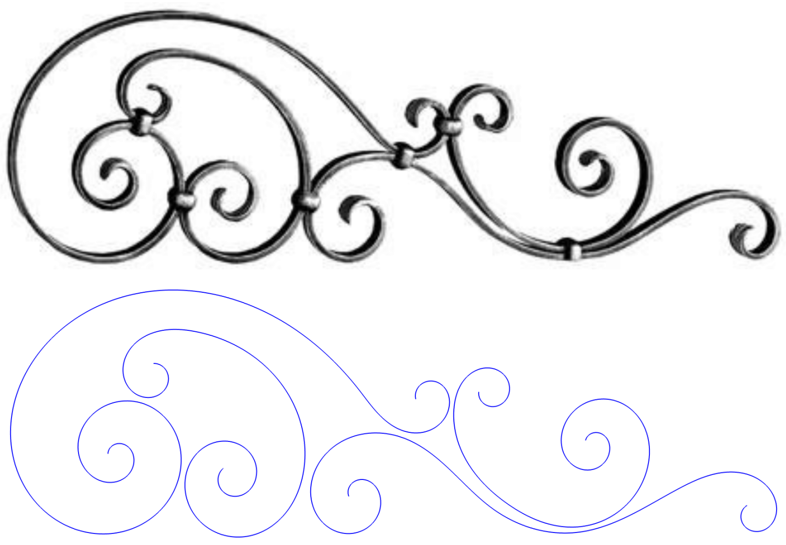


Figure 7.1.: An ornamental wrought iron form, captured with PCCs.

Curvature control is so important in high-quality ("*class-A*") shape design because curves are the basis for creating surfaces, and curvature artifacts lead to bad light reflections. State of the art in industrial design is the use of (piecewise) polynomial curves such as Bézier curves, B-splines, and their numerous variations. They are efficient to compute, their properties are well understood, and many algorithms exist for knot insertion, degree elevation, etc. The first hypothesis, however, precludes curve representations with invisible extra parameters such as NURBS, β-, τ-, or ν-splines; even varying the knot spacing is usually avoided. The use of any such non-graphical parameters has the crucial disadvantage that the curve shape cannot be judged only by looking at the control polygon; but this is exactly the expertise of surface engineers. This is why professional class-A surfacing software like *IcemSurf* still uses exclusively the oldest and most direct surface technique, tensor product Bézier surfaces of degree three up to nine.

The curvature of spline curves is difficult to control. In some cases it is even impossible to obtain the desired curvature practically, i.e., using a sparse control polygon (see Section 7.7). The reason is that, informally speaking, spline control points exert an 'extra pull' on the curve. A B-Spline with a regular n-gon as control polygon still deviates noticeably from a circle; without an extra weight parameter (rational splines) it is impossible to obtain a perfect circle. The real problem, however, is the lack of clear rules for the placement of control points.

Surface engineers need years of experience to master the control point placement intuitively.

7.1.1. Piecewise Clothoid Curves as superior Alternative

The work presented in this chapter is the result of the quest for a curve representation that has no hidden parameters and offers exactly the degrees of freedom that designers need in order to control both the shape and the curvature of a 2D curve. We propose replacing splines by piecewise clothoid curves (PCCs). The heart of the framework is a fast algorithm to compute a PCC from a given (open or closed) sequence of input points. The problem can be stated formally as follows:

> Given 2D points $p_1, \ldots, p_n$, compute clothoid segments $c_1(t), \ldots, c_{n-1}(t)$ with arc lengths $d_1, \ldots, d_{n-1}$ so that $c_1(0) = p_1$, $c_i(0) = c_{i-1}(d_{i-1}) = p_i$ for $i > 1$, and $c_{n-1}(d_{n-1}) = p_n$.

There is no suitable closed-form representation of clothoid curves; they are defined via Fresnel integrals and computing them is impractical [42]. Therefore we use an iterative discrete scheme (see Section 7.3).

7.1.2. Contribution

Our contribution is a unified framework for curvature-controlled curve design with PCCs:

- Fast adaptive clothoid interpolation algorithm
- Curvature simplification by dynamic programming
- Clothoid blending by nonlinear subdivision
- Direct editing of the curvature plot
- Physical interpretation of curvature smoothing
- Different control modes for points: free, constrained tangents, and constrained curvature.

7.1.3. Benefit

Clothoid curves are well known for their aesthetic quality. However, they have never been widely used in shape design, maybe for efficiency reasons, but certainly also because of the lack of suitable design tools. We argue that with PCCs, designers can reach their goal much faster and with an excellent level of control, for example to meet max/min curvature constraints, to limit the curvature variation, and to obtain aesthetically pleasing results. PCCs are much more 'relaxed' than splines; the infamous 'spikes' with unbounded curvature are avoided. Tension can still be added to the curve intentionally by explicitly inserting short segments as it is demonstrated, e.g., in Figures 7.1 and 7.14.

In summary, we show that PCCs are superior to splines in practice: Any spline curve can be approximated by a PCC with a sparse control polygon, but the converse is not true (see Section 7.7).

7.2. Related Work

A variational approach for computing discrete approximations of piecewise clothoid curves was presented in a fairing context in [75]. Like in our algorithm, they refine a given polygon by inserting new points and moving them such that the curvature distribution becomes piecewise linear (direct approach). They also propose an indirect approach, iteratively alternating between curvature estimation at the control points and curvature interpolation at intermediate points. Our method is similar to their direct method, but we take the segment lengths into account. This assures fast convergence also for adjacent segments of greatly varying lengths. This is the key for adding fine details and 'tension' to the clothoids. Clarifying about terminology, our *piecewise clothoid curve* (PCC) is called *discrete clothoid spline* (DCS) in [75]; but since splines are often associated with convolution or blending, we find it more appropriate to call a *clothoid spline* the result of the curvature refinement process presented in Section 7.5.

It was observed already in the 1970s that clothoids are useful for interpolating data points with specified tangent and curvature. As proposed in [66] they can be connected with *linear curvature elements* (linces) that are integrated to yield clothoids. The approach was extended by [68, 74] who developed blending patterns for bi-, tri-, quadrilinces to connect a pair of data points. They used a direct integration method while our method is global and works in a variational setting, interpolating all control points. While their method requires tangents and curvature in every control point, our method guarantees that every two points are connected only by a single clothoid. An advantage of their method is that modifications have only local impact, in contrast to our method where the impact is global; however, we can also achieve the same, since specifying constraints effectively decouples the parts of our curves.

An extension of the clothoid (Cornu spiral) to the generalized Cornu spiral (GCS) is presented in [10]. The curvature is not only linear but rational, so additional degrees of freedom are available; however, they are not visual, and so they are difficult to control by designers. A useful overview of clothoid (Euler spiral) techniques is given in the comprehensive thesis [55]. It includes mathematical foundations, application examples, and also historical background.

The approaches [58, 59] are concerned with fitting a clothoid curve to sketched input data. They use a dynamic programming approach in the curvature domain in order to identify portions of the input curve that can be approximated by a clothoid segment. The fitting algorithm presented in Section 7.8.3 borrows from their dynamic programming approach, but we use it to directly extract suitable PCC control points from the input curve. This simplifies the procedure since we do not have to fit so many clothoid segments; and our PCC does not deviate so much from the input curve since the control points are interpolated.

The method of [14] builds upon the principles of [58] for fitting clothoid curves to sketched input data. A large number of straight line, circle, and clothoid segments is tentatively fitted, and then represented as nodes of a graph from which the best candidate segments are identified using a flow algorithm. The segments found are then optimized in order to meet, thus obtaining a piecewise clothoid curve. This approach has fewer problems than [58] concerning the deviation from the input data, but the method is much more complex than ours; and it does not yield conveniently editable control points.

Another approach building upon [58] is [57]; it is concerned with the related topic of fitting French curves to sketched input data. Also concerned with fitting curves to sketched input data is [88]. They define a so-called *Elasticurve* which is roughly a smooth version of the input curve, represented as lines, parabolas, and arcs.

Like this chapter, [94] is concerned with a clothoid curve representation that is interactively controllable. In contrast to our interpolated control points, they use a control polyline to which a clothoid curve is fitted. A generalization of clothoids to 3D was presented in [16]; their fitting algorithm produces 3D curves where not only curvature but also torsion varies linearly with arc length; however, they are not concerned with high-quality curve design as their main topic.

7.3. Computing Piecewise Clothoid Curves

A piecewise clothoid curve (PCC) consists of various clothoid segments, which may also comprise line segments and circle segments. The segments are joined together such that they are both tangent and curvature continuous in the joints (G^2 continuous).

7.3.1. Discrete PCC Curves

We use a method where a sequence of control points is either specified interactively by the user, or chosen appropriately from a given input curve; then our algorithm constructs clothoid segments joining these points. Clothoids are defined by Fresnel integrals, for which different approximations exist [42]. Instead of computing the parameters of the clothoid segments we use a variational approach generating a polyline with linear discrete curvature, thus approximating the clothoid segment. The approximation error can be made arbitrarily small by iterative refinement.

7.3.2. Iterative Algorithm

The iteration starts with the control polygon, i.e., the polyline through the control points. Then we repeat two alternating steps. First, a new point is inserted between every two points of the polyline; the sequence of points that are inserted between consecutive control points are called a *segment*. Second, the position of all inner segment points (i.e., except the original control points) is optimized. The new position of a point is computed on the perpendicular bisector of its neighbors in such a way that the curvature is the arithmetic mean of the curvatures of its neighbors.

Throughout this chapter, the (discrete) curvature in a point p of a polyline is computed simply as the inverse radius of the circle through p and its two polyline

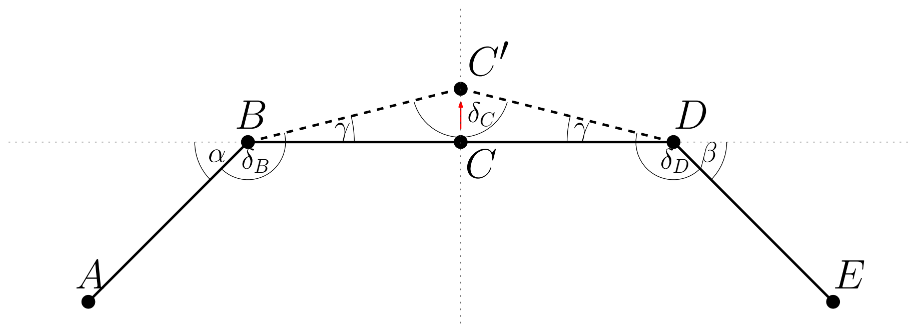


Figure 7.2.: Optimization of point C. Angles α and β are given by the position of the points with respect to the dotted line g through B and D. C' is positioned on the perpendicular bisector of B and D such that δ_C is the arithmetic mean of δ_B and δ_D. γ is used for the calculation.

neighbors. Consequently, five points are involved when optimizing C, namely its direct neighbours and their respective neighbours. The curvature information is transferred from one segment to the next over control points. The second consequence is that inner segment points quickly converge to equal spacing and to linear curvature distribution, thus obtaining a discrete clothoid.

7.3.3. Point Positioning

Figure 7.2 illustrates the position computation. In order to insert (or update) point C with neighbours B and D, their respective neighbors A and E are considered. Without loss of generality we use the normalized configuration where B lies in $(-1,0)^T$ and D in $(1,0)^T$.

We use the following notation:

- let X be one of the five control points, and X_l his left and X_r his right neighbor
- δ_X is the angle between $\overrightarrow{XX_l}$ and $\overrightarrow{XX_r}$
- κ_X is the discrete curvature in point X
- g is the line through B and D
- α is the angle between g and $\overrightarrow{BA}$
- β is the angle between g and $\overrightarrow{DE}$
- γ is the angle between g and $\overrightarrow{BC}$

Two conditions must be fulfilled for C:

- C must lie on the perpendicular bisector between B and D.
- κ_C must be the arithmetic mean of κ_B and κ_D

As more and more points are inserted, the polyline gets refined and the angles between segments approach π, and angles α and β approach 0. In the following formulae we use some simplifications whose errors converge to 0 when α and β approach 0. From the conditions we obtain:

$$\kappa_C = \frac{1}{2}(\kappa_B + \kappa_D). \tag{7.1}$$

We approximate the discrete curvature of X by

$$\kappa_X = 2\frac{\pi - \delta_X}{|\overrightarrow{XX_l}| + |\overrightarrow{XX_r}|} \tag{7.2}$$

as proposed in [15]. Then we substitute

$$\begin{aligned} \delta_B &= \pi - \alpha + \gamma && (7.3)\\ \delta_C &= \pi - 2\gamma && (7.4)\\ \delta_D &= \pi - \beta + \gamma. && (7.5) \end{aligned}$$

Since $|\overrightarrow{BD}| = 2$ we approximate $|\overrightarrow{BC}|$ and $|\overrightarrow{CD}|$ with 1, because when α and β approach 0, γ also approaches 0. Solving the resulting formula for γ finally yields

$$\gamma = \frac{\beta(|\overrightarrow{BA}| + 1) + \alpha(|\overrightarrow{DE}| + 1)}{2|\overrightarrow{BA}||\overrightarrow{DE}| + 3(|\overrightarrow{BA}| + |\overrightarrow{DE}|) + 4}. \tag{7.6}$$

Point C is now inserted on the perpendicular bisector between B and D in distance $\tan\gamma$ from g.

When inserting a point next to an endpoint the problem is that a neighbor is missing on one side, i.e., A with α, or E with β. We need to specify an additional constraint to obtain a unique solution. We can enforce either a tangent or a curvature constraint, which are presented in Section 7.4. For endpoints the usual default is a curvature constraint that enforces zero curvature.

In the case of evenly spaced points this formula is identical to the discrete fairing approach proposed by [75]. Their *discrete clothoid spline* (DCS) is an evenly spaced

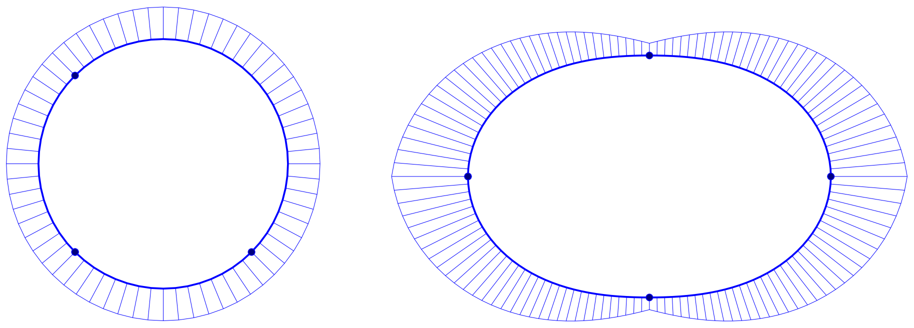


Figure 7.3.: PCC configurations. The closed PCC through three control points always becomes a circle. Four control points in an 'elliptical' configuration do not lead to an ellipse, but to an ellipse-like ovoid with piecewise linear curvature.

polyline with the condition that the curvature at each point is the average of its neighbor curvatures:

$$\kappa_i = (\kappa_{i-1} + \kappa_{i+1})/2 \; .$$

Thus, the curvature varies linearly, and the resulting curve must be a clothoid. Note that κ_i is the discrete curvature, i.e., the inverse radius of the circle through three points p_{i-1}, p_i, p_{i+1}. It can be computed using the inverse of the well-known triangle circumcircle formula $\kappa = 1/r = 4A/abc$ of a triangle with edge lengths a, b, c and (signed) area A.

7.3.4. Properties

PCCs have some nice properties for designers. The closed PCC through three points always is a circle (see Fig. 7.3); adding further control points is required only to define the deviation from a circle shape.

The position of a control point influences the PCC globally, meaning that every part of the PCC changes when moving a single control point. The influence of the control point, though, is heavily damped, as shown in Fig. 7.4. The damping effect can also be amplified by placing control points very closely together; moving a point on one side then has little effect on the curve on the other (Fig. 7.5). This is similar to a tangent constraint.

Inserting additional control points on a PCC neither changes the curve nor its curvature. This is extremely helpful in order to add fine detail, since a few close points can be inserted to 'nail down' some part of the curve by exploiting the

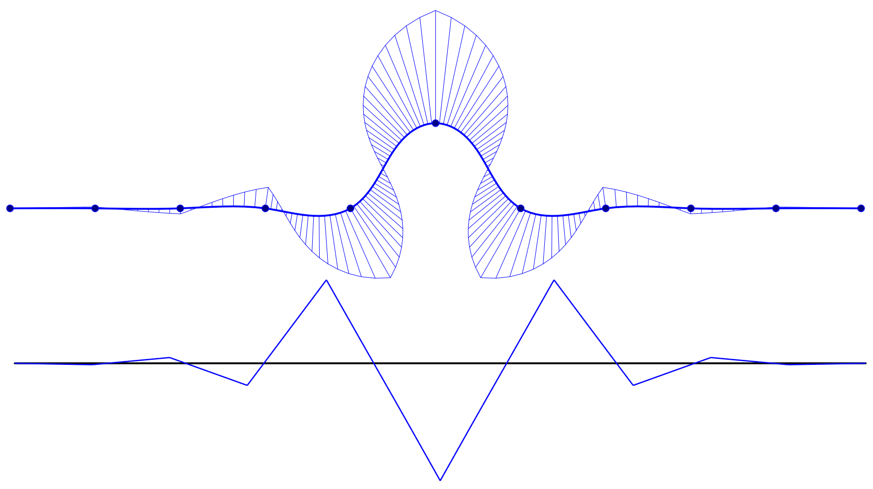


Figure 7.4.: Damping of PCC curves. Top: The control points of the PCC are uniformly distributed, and the middle one is moved upwards by the same distance. Bottom: Curvature of the PCC.

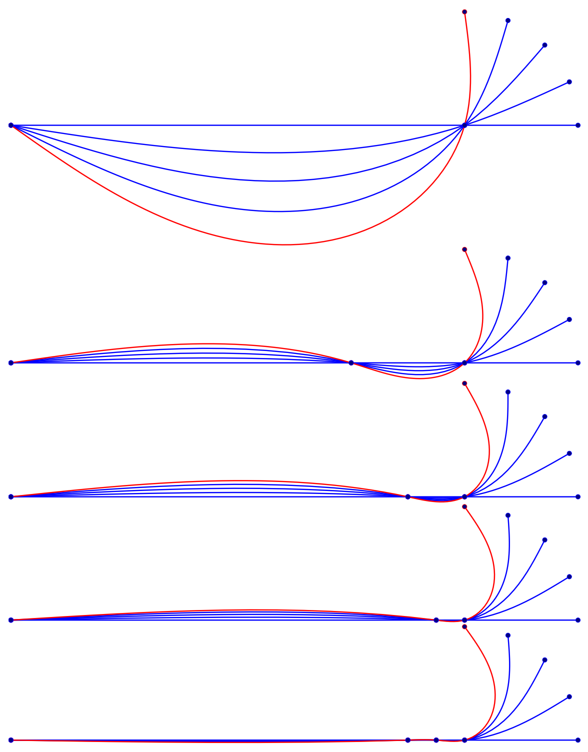


Figure 7.5.: Damping effect of closely spaced control points.

damping effect. One consequence of the piecewise linear curvature behaviour is that curvature maxima *always* lie in the control points (in contrast to splines!). This makes PCCs much easier to understand and control by designers.

7.4. Constraints on Tangents and Curvature

To speed up the design process, tangent or curvature constraints can be defined for each control point:

- A tangent constraint defines the tangent of an adjacent clothoid segment.
- A curvature constraint defines the curvature of an adjacent clothoid segment.

The constraints can be the same or different on both sides, e.g., different tangents produce a corner (G^0). It is also possible to define a tangent constraint on one side and a curvature constraint on the other, to maintain G^1. Note, however, that prescribing the same curvature on both sides will in general still lead to a corner (G^0). This mode is still valuable since the designer can move the control point to make the tangents gradually more collinear, to trade shape against curvature (see Fig. 7.6).

7.4.1. Tangent Constraints

Formula (7.6) can be easily modified to account for a tangent constraint (c.f. Fig. 7.2): B is the point with the tangent constraint and neighbour A is discarded for the computation. Instead, angle α is set to the angle between the prescribed tangent and the line g through B and D. Length $|\overrightarrow{BA}|$ can be set to zero because for the computation we consider a point on the tangent instead of A, and the tangent condition has more influence the closer the point is to B.

7.4.2. Curvature Constraints

Let again B be the point with the curvature constraint. δ_B and κ_B are not relevant now, but we can imagine a virtual segment $\overrightarrow{BA}$ and choose it in such a way that

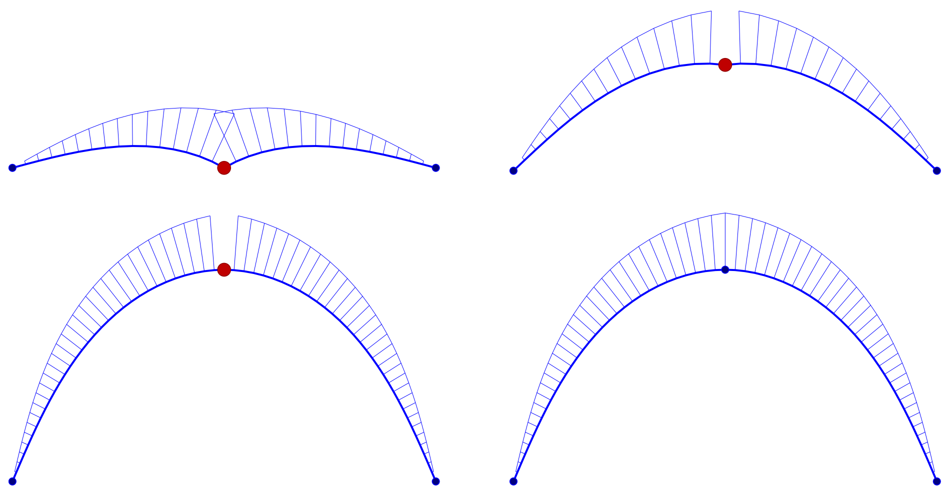


Figure 7.6.: Trading shape against curvature. With three points in a row, a curvature constraint (marked in red) leads to a corner. It can be alleviated by moving the control point, changing shape until eventually the designer chooses to release the curvature constraint as it is sufficiently met. The curvature plot of all four curves is identical.

κ_B has the desired curvature value. If $|\overrightarrow{BA}| = 1$, then

$$\kappa_B \approx \frac{1}{\frac{1}{2}\sec\frac{\delta_B}{2}} = 2\cos\frac{\delta_B}{2} = 2\sin\left(\frac{\pi}{2} - \frac{\delta_B}{2}\right) \approx \pi - \delta_B \tag{7.7}$$

for small values of $\pi - \delta_B$. From this follows:

$$\gamma = \frac{2\,\beta + \kappa_B\,(|\overrightarrow{DE}| + 1)}{4\,|\overrightarrow{DE}| + 6} \tag{7.8}$$

At last, when we only have two endpoints and insert a point between them, on both sides a neighbor is missing. Taking $|\overrightarrow{DE}| = 1$ and $\kappa_D = \pi - \delta_D$ then leads to

$$\gamma = \frac{1}{4}(\kappa_B + \kappa_D)\ . \tag{7.9}$$

7.4.3. Restoring G^2 Continuity for collinear Tangents

A control point P with collinear tangents still achieves only G^1 continuity since in general, the curvatures do not match anymore. To restore G^2, another control

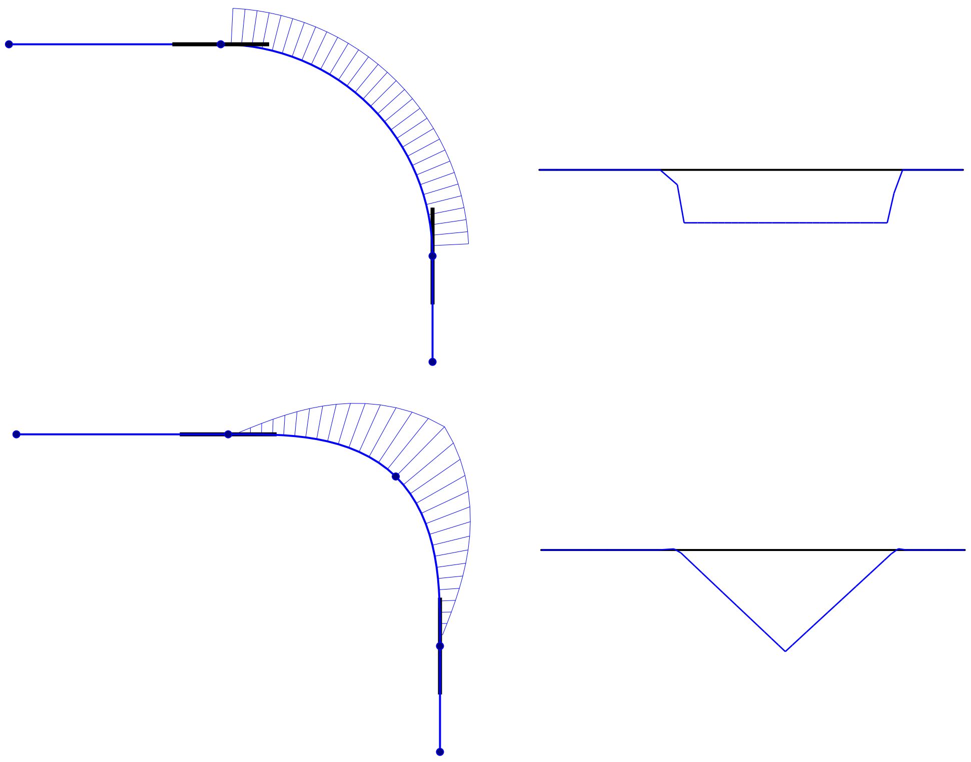


Figure 7.7.: Restoring G^2 continuity. Top: A 90° configuration with straight sides and a rounded corner. The straightness of the sides is ensured with tangent constraints. The constraints permit continuous curvature, as can be seen in the curvature plot. Note: Because only the discrete curvature is measured, the spot with the curvature jump deviates from a vertical line. Bottom: An additional inserted point, moved to the right spot restores continuous curvature.

point can be inserted on one side of P and moved such that the curvatures at P match on both sides; its position is not unique, so there is some design freedom.

Especially useful is the case of G^2 continuous rounded corners (see Fig. 7.7). First, control points are set in the positions for the start and end of the rounding. Then the tangents for the rounding are specified. At last, an additional control point is inserted in such a position that the joints at both control points simultaneously become G^2. The position of this point is unique and can, thus, be computed automatically, e.g., iteratively.

7.4.4. Direct Curvature Control

The properties presented so far suggest a very straightforward workflow for curve reconstruction with PCCs: Since control points are interpolated, and inserting a point does not change the curve, the designer can simply keep adding points along the desired contour. Curvature, however, is a very sensitive measure. Obtaining a good curvature profile is a fiddling task. We now present a method where the designer can *directly edit the curvature profile* to obtain very rapidly the profile shown in the bottom.

The idea is to simply move the control point perpendicular to the line through its neighbours. Consider three points A, B, C and a desired target curvature κ_t for B. We use a simple linear estimate for the new position of B. Let κ_B be the discrete curvature for B, x the perpendicular projection of B to $\overrightarrow{AC}$, and d the distance between B and x. When x is positioned more towards the center of A and C, then the curvature does vary less when B is moved perpendicular to $\overrightarrow{AC}$. So we calculate a scalar factor λ for the relative position of x with $\lambda = 2\,\min(\,\|A-x\|, \|C-x\|\,)\;/\;\|C-A\|$.

The estimate for the needed offset distance is then $d_o = \lambda \cdot (\kappa_t - \kappa_B) \cdot \|C-A\|^2 \cdot c$ with the constant $c = \frac{1}{11}$ that was determined by experiment. Point B is moved by d_o along the perpendicular to line $\overrightarrow{AC}$. Then the whole PCC is recomputed, and the process is repeated until the desired accuracy is achieved, or further movement does not bring the curvature closer to the target curvature anymore, e.g., in case a high curvature was desired but the neighbours are too far apart.

7.5. Curvature Blending and Clothoid Splines

7.5.1. Eliminating Curvature Spikes

The curvature of a PCC is piecewise linear but not bounded; there can be arbitrarily steep spikes. One possibility to get rid of a curvature spike in a control point P is to first insert two new control points close to P on either side, which changes neither the curve nor its curvature. Then removing P does change the curve, but only slightly; and the curvature-spike is cut off.

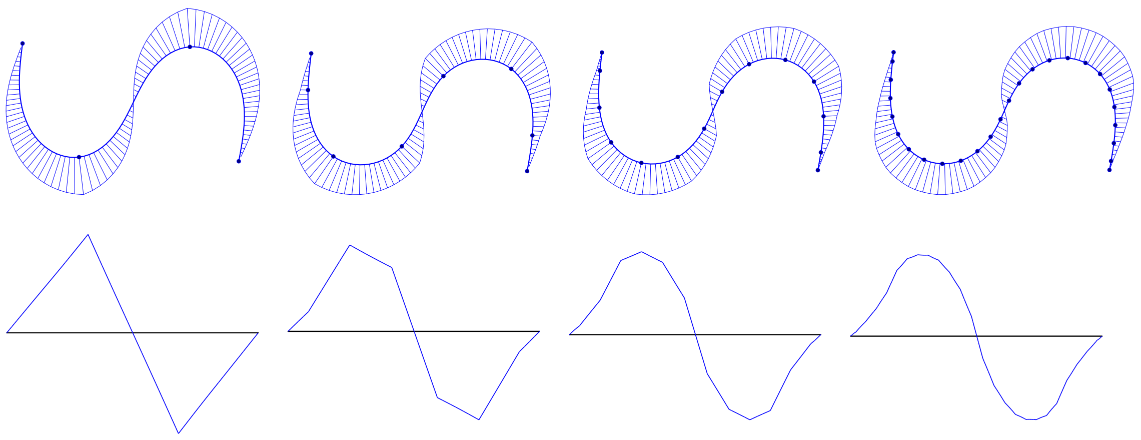


Figure 7.8.: Construction of a clothoid spline. Repeated insertion of new points and removal of old points.

7.5.2. Clothoid Splines: Approaching the Limit

Our method for iterating this scheme is inspired by Chaikin's corner cutting method for obtaining quadratic B-Spline curves [20]. The method from the previous Section 7.5.1 can be repeated arbitrarily often.

For closed PCC curves, new control points are inserted on each clothoid segment at $\frac{1}{4}$ and $\frac{3}{4}$ of the arc length. Then the old control points are removed and the PCC is recalculated. This leads to a PCC with twice the number of control points. This process is repeated for a certain number of iterations. The control points of the last step are taken as the points of a "*clothoid spline*".

For open curves, the process is nearly the same. However, the start point and the end point are never removed. In the first and in the last segment, only a single new control point is inserted at position $\frac{1}{2}$ in the first iteration; otherwise the refined points would accumulate at the start and at the end. Fig. 7.8 illustrates this construction of a clothoid spline with G^3 continuity.

7.6. A physical Interpretation of Clothoids

A vast number of different curve representations has been proposed in CAGD over the last 60 years. With any new representation the question must be answered how suitable it is for the targeted purpose. For high-end curve design, it should

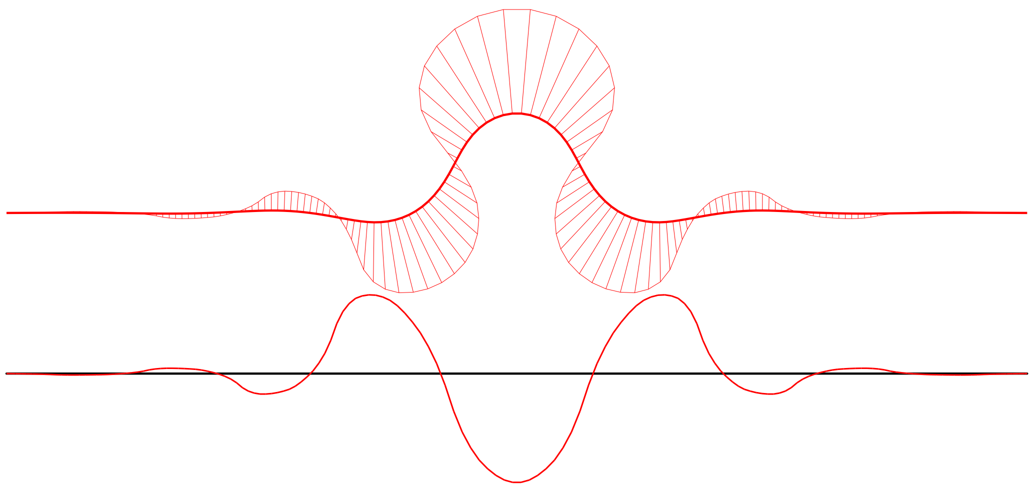


Figure 7.9.: PCC spline. The control points are the same as in Fig. 7.4. There is hardly any difference between the curves, but a great difference between the curvatures: The PCC spline is G^3.

make designing aesthetic curves as simple as possible. Capturing the notion of aesthetics is difficult, however, which is witnessed by the lack of a common mathematical definition of *class A quality*.

We have approached the problem from a different side. The reader is encouraged to try a very simple physical experiment, namely designing a 'smooth' curve with fixed-length segments, e.g., *Kapla* bricks (Fig. 7.10). It will turn out sooner or later that instead of looking 'along' the curve, the obvious thing to do is focusing on the gap angles between the bricks, and to balance them; an uneven gap distribution is deemed ugly *by anybody*.

Since the gaps are small, attaching normal rods to the segments shows problems more clearly. Equalizing the gap angles can then be seen as equalizing the distances between the end points of consecutive rods. Physically, this could be accomplished by attaching springs on the end points; and for symmetry reasons, two such rods should be attached, one on eiter side of the segment.

This leads to a variational energy minimization problem: A *spring element* is made out of five points A, B, C, D, E connected by line segments with two normal rods that are connected by six springs with rest length zero (Fig. 7.11). The position of $C(t)$ is optimized on the perpendicular bisector between B and D, the other points are fixed. Finding t for which the spring energy $E_s(t)$ is minimal is a convex univariate problem.

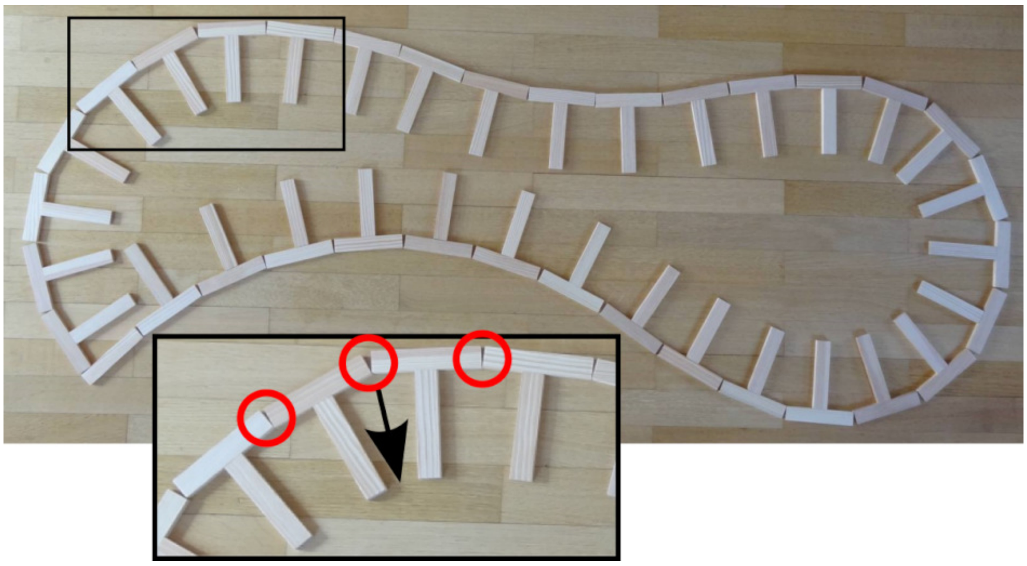


Figure 7.10.: Design experiment: Aesthetic improvement of a curve. In this situation, virtually anybody will move the bricks down to equalize the gap angles. Normal rods show the angle distribution more clearly.

The spring energy is proportional to the square of the distance, but the derivative is not linear in t. However, since the energy is convex, the optimum is easily reached by a few Newton iterations. Mimicking the manual method, we turned this into a curve optimization scheme by keeping some of the input points fixed while optimizing the free points. The implementation revealed a surprise: It converged rapidly to a curve with a linear curvature profile between the fixed points – obviously a sequence of clothoids! So the spring elements provide a clear, intuitive physical interpretation of curve design with clothoids that can be easily grasped by artists and designers.

7.6.1. Analytical Solution for Infinite Rod Lengths

The length of the normal rods is a free parameter. We made the observation that the optimization yields better results with longer rods, which led us to examining the case of infinite rod length. As shown in the following, this yields the same formula as in Sec. 7.3.

The longer the rods, the smaller are the segments in comparison. So instead of infinite rod length, we consider rods of fixed (unit) length and zero length

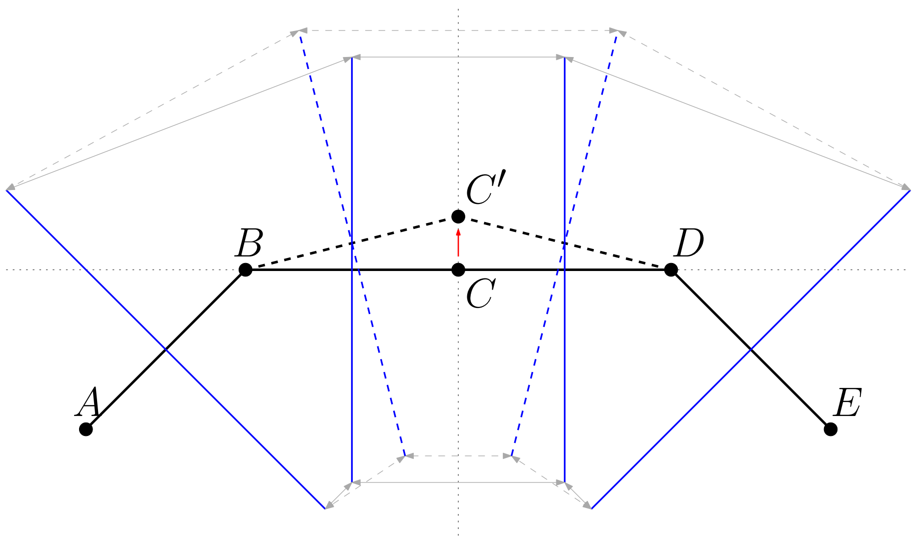


Figure 7.11.: A spring element with fife points, four segments, eight normal rods and six springs connecting the end points of the rods. The position of the center point is spring-optimized along the perpendicular bisector of the line segment connecting its neighbours.

segments. The end points of the rods lie on the unit circle; let σ be the angle between two rods, then the force $f(\sigma)$ is the squared distance of their endpoints:

$$f(\sigma) = 2\,\mathrm{chord}^2\sigma = 8\sin^2\frac{\sigma}{2}. \tag{7.10}$$

For small values of σ the sine function is approximately the identity. This yields a simplified version f_s of the force:

$$f_s(\sigma) = 2\,\sigma^2. \tag{7.11}$$

Figs. 7.2 and 7.11 show that the angle between the 1st and the 2nd rod is $\alpha - \gamma$, between the 2nd and the 3rd it is 2γ, and between the 3rd and the 4th it is $\beta - \gamma$. For the spring element with points A, B, C, D, E we have a total (simplified) force f_t of

$$f_t = f_s(\alpha - \gamma) + f_s(2\gamma) + f_s(\beta - \gamma) \tag{7.12}$$

In order to find the value of γ with the minimal force in the spring elements we differentiate f_t, set it to zero, and solve for γ:

$$f_t' \quad = \quad -4\alpha + 24\gamma - 4\beta \tag{7.13}$$

$$f_t' \quad = \quad 0 \quad \Rightarrow \quad \gamma = \frac{1}{6}\alpha + \frac{1}{6}\beta \tag{7.14}$$

This is indeed the same formula as in Eq. (7.6) when $|\overrightarrow{BA}|$ and $|\overrightarrow{DE}|$ are equal to one. So with evenly spaced points, the two approaches yield the same results.

7.7. Comparison with B-spline Curves

As mentioned in the Introduction, a fair comparison from the design point of view can take into account only spline types without non-visual parameters (knots, weights). Comparing, for example, PCCs to uniform cubic B-splines, PCCs have the obvious advantages that control points are interpolated instead of approximated, that point insertion does not change the curve, and that circular arcs can be reproduced.

A more serious issue, though, is that despite the variation diminishing property and their C^2 continuity, the curvature even of higher-degree polynomial splines is unbounded and hard to control in practice. Uneven control point spacing is likely to result in unexpected curvature maxima, and even spikes. With PCCs, the curvature can clearly be (locally) maximal only in the control points, which is a clear and practical placement rule.

We claim that for all practical design tasks, PCCs are in fact ***superior*** to splines: While it is easy for PCCs to realize the 'extra pull' of spline curves, it is impossible for splines to reproduce the perfectly clean curvature profile of PCCs. For controlling curvature (class-A design), PCCs may even be seen as the *optimal* curve representation because any desired curvature profile can efficiently be approximated using linear segments.

To quantify these claims, Fig. 7.12 shows that a PCC with three times as many control points can nicely reproduce the curvature artifacts of a spline; and that by doubling the number of spline control points, the artefacts are only damped but their frequency is doubled as well. Note that curvature artifacts are unavoidable with splines, as shown by Augsdoerfer et al. [12] who treat the subject in considerable depth.

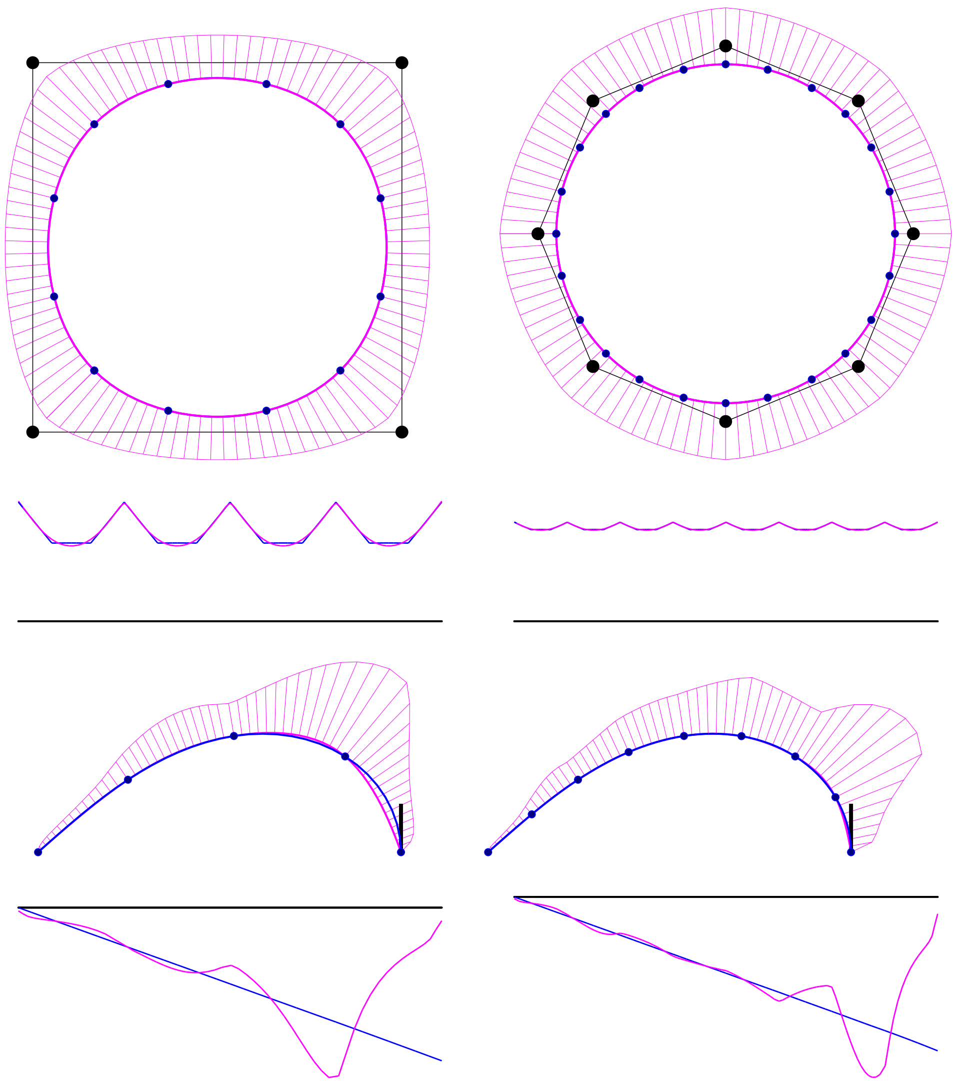


Figure 7.12.: Curvature artifacts of spline curves. Top: A uniform cubic B-spline (magenta) with four control points can be approximated by a PCC with twelve control points - including the curvature artifacts (left). Completely avoiding the artifacts is not possible (right). Bottom: A single clothoid segment is approximated by a spline. Despite regular spacing the resulting curvature is simply inacceptable.

7.8. Results and Applications

7.8.1. Interactive Shape Design with PCCs

An important property for a curve representation is how effective and accurate designers can work with them. The proposed PCCs have some nice properties for the design process:

- Reasonable approximations can be computed quite fast also for larger numbers of control points.
- Insertion of new control points neither changes the curve nor does it affect other control points.
- Redundant control points (with the same curvature slope on both sides) can be removed without affecting the curve.
- Tangents or curvature constraints can be specified for all control points.

7.8.2. Curve Design Rules for PCCs

We demonstrate the typical design flow with a particularly challenging example, the insertion of details on a larger curve with very smooth curvature (Fig. 7.13):

1. The designer draws a few spacious control points to define a shape with smoothly flowing curvature.
2. Detail zones are isolated by inserting pairs of control points with tangent constraints.
3. Editing the zones does not affect the rest of the curve, but reduces continuity to C^1 in these points.
4. Finally, G^2 continuity can be restored with the technique explained in Sec. 7.4.3 (c.f. Fig. 7.7).

We claim that PCCs are especially well suited for high-quality curve design. This is supported by empirical evidence since the characteristic curves of many existing high-quality shapes can be captured using only very few control points, as illustrated by Figures 7.1 and 7.14. And even more important is that the control points can be obtained using a clear coarse-to-fine recipe.

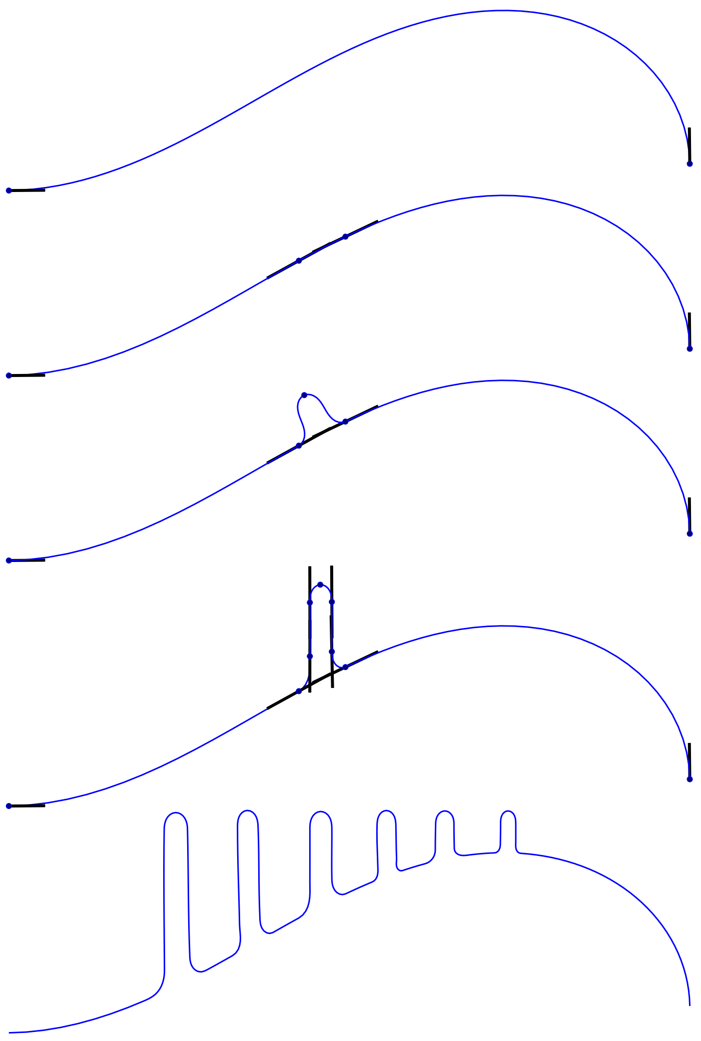


Figure 7.13.: Inserting U-shapes on a large smooth curve segement without affecting the rest of the curve. Top-to-bottom: Original curve. Insertion of two control points with fixed tangents, then local modification on isolated part. The curvature of the seemably fixed-radius blend and in the tips can be controlled by editing the curvature plot.

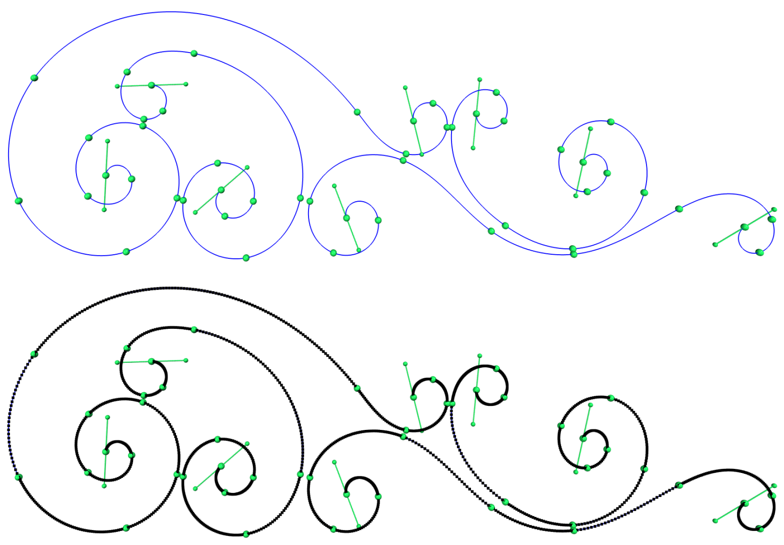


Figure 7.14.: Shapes of a wrought iron captured with PCCs, c.f. Figure 7.1. The top shows the PCC with its control-points and tangent constraints at the end-points. The bottom additionally shows the calculated curve-points (in black).

7.8.3. Computing a PCC for a sampled Input Curve

Our second use case is fitting a PCC to a given densely sampled input curve (polyline). The automatic algorithm presented in this section uses dynamic programming to partition the curvature profile of the input curve in order to find ideal control point positions. The idea to use dynamic programming was borrowed from [58]; the algorithm chooses points from the input curve which are directly used as PCC control points. The PCC shall follow the input curve very closely and should have a sparse adaptive control polygon.

To penalize the number of control points, each PCC segment is associated with a cost; and to enforce the similarity of both curvature profiles, the deviation of the PCC curvature from the input curvature is penalized as well. Note that for demonstration purposes we do *not* even include an explicit cost for the geometric distance between the curves. In all of our examples it was sufficient that (a) the PCC control points are drawn from the input polygon, and (b) the curvature profiles are very similar; note that 2D curves are uniquely defined by their curvature (up to rigid transformations).

In the following, let a and b denote the arclength parameters of two points $P(a)$,

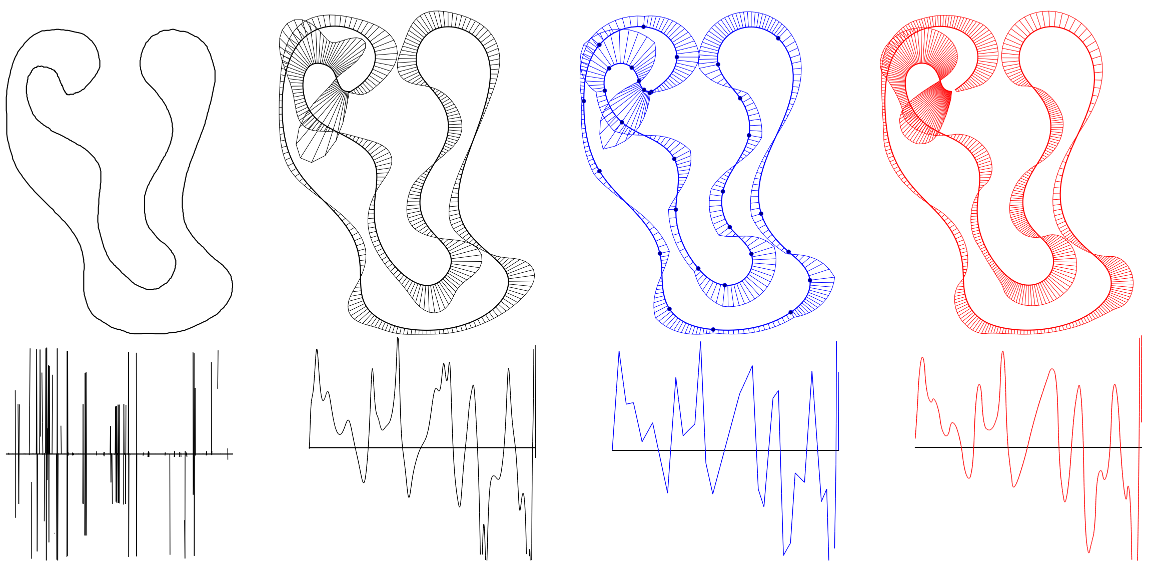


Figure 7.15.: Fitting a PCC to input data. From left to right: 1) Original curve, 2) Smoothed curve, 3) PCC fitted to the smoothed curve using dynamic programming as described in Section 7.8.3. Big dots denote control points, small dots are points inserted by the PCC algorithm. 4) Clothoid spline with smooth curvature profile as described in Section 7.5.2. The input curve is taken from [14].

$P(b)$ of the input curve (a discrete polyline), and $P(a,b)$ is the curve segment between a and b. We define a cost matrix M where each entry $M(a,b)$ is the minimal cost of placing control points at a and b, and anywhere inside $P(a,b)$. It is defined recursively in the typical dynamic programming fashion:

$$M(a,b) = \min_{a<k<b} \{E_d(a,b) + \lambda E_i(a,b),\ M(a,k) + M(k,b)\} \tag{7.15}$$

The first term is the cost when no other control point is used between a and b, and the second term applies when splitting up the curve by inserting another control point at any possible parameter $a < k < b$ of a polyline point yields (recursively) even smaller cost. Concerning the first term, E_d penalizes the deviation of the PCC curvature from the input curvature (discrete integral of absolute differences). We define it as

$$E_d(a,b) = \sum_{c \in P(a,b)} f(c)\, \|\kappa(a) - \kappa(c) + l_s(c-a)\|^{e_d} \tag{7.16}$$

where $l_s = \frac{\kappa(b)-\kappa(a)}{b-a}$ is the slope of the curvature function in this interval and $f(c) = \frac{1}{2}(c_+ - c_-)$ is the local strip width with respect to the previous and next points c_-, c_+ of the polyline. E_i penalizes short segments:

$$E_i(a,b) = \left(\frac{d}{b-a}\right)^{e_i} \tag{7.17}$$

where d is the overall length of the polyline. Parameters λ, e_d, e_i can be tuned by the user in order to specify the different trade-offs.

The matrix M is computed in a bottom-up fashion, eventually yielding $M(0,d)$ as the cost of the best possible selection of polyline points as PCC control points to match the input curve.

An example of the fitting process is shown in Fig. 7.15. The input curve is smoothed before the fitting starts in order to achieve better results.

7.9. Conclusion

We have presented in this chapter a framework for the design of high-quality curves. To summarize, piecewise clothoid curves (PCCs) have the following key advantages over splines:

- Direct editing: The control polygon is interpolated, rather than just approximated.
- Insertion invariance: Inserting control points does not change the curve.
- Curvature extrema: The curvature is (locally) maximal or minimal only in the control points.
- Predictability: Control points are needed only in order to deviate from linear (circular) curvature.

Intuitive methods were presented for constraining the PCC to specific tangents or curvature values, and for automatically improving the piecewise linear curvature profile, eventually resulting in a 'clothoid spline' with smooth curvature (G^3). It was shown that a given input curve can be efficiently converted to a PCC, and controlling curvature is possible even to the point of directly editing the curvature profile.

For designers, this can be intuitively summarized as *"PCCs are class-A by default"*; a relaxed type of curve for which introducing tension requires specific effort.

7.10. Future Work

An important area for future research is a practical method for designing high-quality space curves, where not only curvature but also torsion must be controlled; for a 3D generalization of clothoids, Ben-Haim et al.[16] propose varying also the torsion linearly. The main goal, however, will be to find also a surface representation that is acknowledged by practitioners as being 'class-A by default'.

8. Evaluation of the Techniques

8.1. Modeling Capabilities

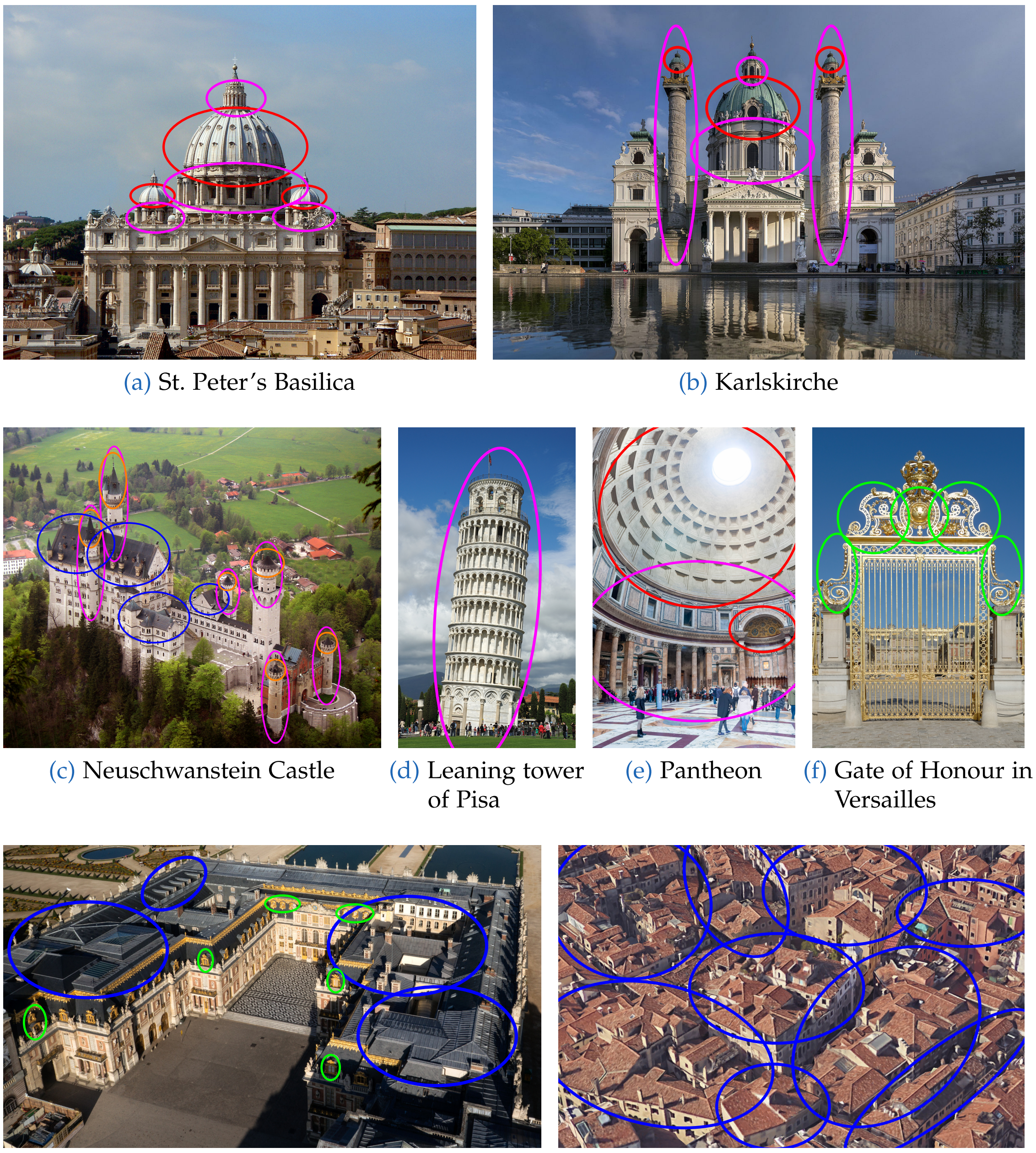
The presented methods in the case studies (see Chapters 5, 6, and 7) show how different parts of historical buildings can be modeled via new procedural modeling techniques. This was previously hardly possible due to the limited expressiveness of previous methods. While parts of buildings could already be modeled very efficiently with previous methods, others could not. This section shows places where the new methods can be used, looking at the examples given in the introduction in Figure 1.2.

The respective parts where the new techniques can be used are circled in Figure 8.1. There a color coding is used to identify which technique can be used in this part. The coding is as follows:

- **Blue:** possible usage of the abstract building and roof model from the case study in Chapter 5.
- **Red:** possible usage of spherical coordinate systems from the case study in Chapter 6.
- **Magenta:** possible usage of cylindrical coordinate systems from the case study in Chapter 6.
- **Orange:** possible usage of conical coordinate systems from the case study in Chapter 6.
- **Green:** possible usage of piecewise clothoid curves for shape modeling from the case study in Chapter 7.

The examples from Figure 8.1 have complex geometry that is representative of those in historic buildings. In particular:

- *St. Peter's Basilica:* In this example the dome in the middle and the little cupolas in front give a good opportunity to be modeled via spherical coordinate systems from the case study in Chapter 6. Beneath a dome or a



(a) St. Peter's Basilica

(b) Karlskirche

(c) Neuschwanstein Castle

(d) Leaning tower of Pisa

(e) Pantheon

(f) Gate of Honour in Versailles

(g) Palace of Versailles

(h) Aerial view of Venice

Figure 8.1.: Complex historic architecture from Figure 1.2. The parts that can now be modeled with the new modeling techniques from the case studies are circled. Images taken from [70, 54, 95, 17, 91, 22, 89] and ©2018 Google, Map data ©2018 Google.

cupola is usually a circularly arranged set of supporting structural elements and walls. This is also the case here and cylindrical coordinate systems from the case study in Chapter 6 are suited to model this sections. Additionally, on top of the dome there is a little structure that is also arranged circularly.

- *Karlskirche:* This example is very similar to the St. Peter's Basilica. There is a dome in the middle, suitable for spherical coordinate systems, and circularly arranged structures beneath and above. Additionally this example shows two big pillars aside the main entrance. If they should be modeled with high detail cylindrical coordinate systems could be used here. Small domes conclude the pillars on the top.
- *Neuschwanstein Castle:* In this castle a multitude of towers are present that can be modeled with the techniques from Chapter 6. Their walls can be modeled with circular coordinate systems, and their roofs are suited to be modeled by conical coordinate systems (which was indeed shown in Figure 6.11). Additionally, the castle has some complex roof parts which could be modeled by the technique described in Chapter 5.
- *Leaning tower of Pisa:* The tower is one big cylindrically arranged structure and therefore its elements are obviously suited to be arranged according to a cylindrical coordinate system as described in Chapter 6. Additionally the round arches on the outside might also be modeled according to automatically generated cylindrical coordinate systems as described in Chapter 6.6.1.
- *Pantheon:* The main part of the pantheon is clearly the huge dome in the middle whose coffering in the inside follows a spherical coordinate system. The ground floor consists of the cylindrically arranged structural elements that support the dome. Additionally some smaller arches and cupolas are present in the ground floor. All of those elements are suited for the techniques from Chapter 6.
- *Gate of Honour in Versailles:* The decorative elements of the gate pose typical ornamental forms that are very well suited to be modeled with picewise clothoid curves as shown in 7. The spiraling forms have curvature properties that can adequately be reproduced with picewise clothoid curves.
- *Palace of Versailles:* The different wings and building parts of the palace can be modeled with the solid building primitives described in Chapter 5. The primitives can create the base geometry for the facades, but also the complex roof geometry of the palace. The base facade geometry can then afterwards be modeled with conventional split grammar techniques similar to the example in Figure 5.1. The palace also shows some decorative

elements that can be modeled with piecewise clothoid curves as described in Chapter 7. In Figure 8.1g some of the more complex roof constellations of the palace are marked together with some of the decorative dormers on top of the facade.
- *Venice roof landscape:* Venice has a very fine-grained roof landscaped consisting of many small house and roof parts. These parts can be represented by solid building primitives as described in Chapter 5. There are many connections between the individual parts that result in a variety of different roof geometries (some of them are depicted in Figure 5.3). Figure 8.1g shows a section of the roof landscape in Venice. All of the roofs can be modeled with the mentioned technique. Some of the building blocks with connected roofs are marked with circles.

8.2. Used Domain Specific Methods

This section is going to briefly discuss how the presented procedural modeling concepts compare, and which domain specific methods they are using.

In Table 8.1 the methodologies used are listed. As can be seen, a lot of procedural modeling approaches use the split grammar method. And they are implementing it as an external DSL as a Production Rule System.

L-Systems are used for generating the street layout for cities. Only *Procedural Modeling of Cities* is really concerned with this task. The other papers assume the street layout as a given.

Table 8.2 compares which geometric representation and techniques the methods use. The usual geometry representation is triangle meshes. However, often constructive solid geometry (CSG) is used, and therefore geometry can be represented internally as solids. For curved geometry, point interpolation is often used.

Table 8.3 shows which kind of objects can be modeled with a method. Most of the methods concentrate on modeling facades - this is also the area where split grammars are most useful. Many papers show results of complete buildings with facades, roofs, their surrounding lot, embedded into a street layout in a virtual city. But their method is not necessarily concerned with generating all these components, but only one or a few.

Paper	Split grammar	L-system	Scripting language	DSL - internal	DSL - external	DSL - Production Rule System	GUI
Procedural Modeling of Cities	-	x	-	-	x	x	-
Instant Architecture	x	-	-	-	x	x	-
Procedural Modeling of Buildings	x	-	-	-	x	x	-
Advanced Procedural Modeling of Architecture	x	-	o	-	x	x	-
Shape Grammars on Convex Polyhedra	x	-	o	o	x	x	-
Procedural Architecture using Deformation-Aware Split Grammars	x	-	o	o	x	x	-
Generalized Use of Non-Terminal Symbols for Procedural Modeling	x	-	x	x	x	x	-
Creating Procedural Window Building Blocks using the Generative Fact Labeling Method	-	-	-	-	x	-	-
Component-Based Modeling of Complete Buildings	x	-	x	o	-	-	-
Constructive Roofs from Solid Building Primitives	-	-	-	-	x	-	-
Procedural modeling of Architecture with Round Geometry	x	-	x	x	-	x	-
Curvature-controlled Curve Editing using Piecewise Clothoid Curves	-	-	-	-	-	-	x

Table 8.1.: Comparison of used (domain specific) techniques

Paper	Triangle mesh	Solids (CSG)	Point interpolation
Procedural Modeling of Cities	x	-	-
Instant Architecture	x	-	-
Procedural Modeling of Buildings	x	-	-
Advanced Procedural Modeling of Architecture	x	x	-
Shape Grammars on Convex Polyhedra	-	x	-
Procedural Architecture using Deformation-Aware Split Grammars	-	x	x
Generalized Use of Non-Terminal Symbols for Procedural Modeling	x	-	x
Creating Procedural Window Building Blocks using the Generative Fact Labeling Method	-	x	-
Component-Based Modeling of Complete Buildings	x	x	-
Constructive Roofs from Solid Building Primitives	-	x	-
Procedural modeling of Architecture with Round Geometry	-	x	-
Curvature-controlled Curve Editing using Piecewise Clothoid Curves	-	-	x

Table 8.2.: Geometric primitives used by the techniques

Paper	Streets	Building lots	Building shells	Facades	Roofs	Interiors	Ornaments
Procedural Modeling of Cities	x	x	x	x	-	-	-
Instant Architecture	-	-	x	x	o	-	-
Procedural Modeling of Buildings	-	x	x	x	x	-	-
Advanced Procedural Modeling of Architecture	-	x	x	x	x	-	-
Shape Grammars on Convex Polyhedra	-	-	x	x	o	-	-
Procedural Architecture using Deformation-Aware Split Grammars	-	-	x	x	o	-	-
Generalized Use of Non-Terminal Symbols for Procedural Modeling	-	-	x	x	-	-	-
Creating Procedural Window Building Blocks using the Generative Fact Labeling Method	-	-	-	x	-	-	-
Component-Based Modeling of Complete Buildings	-	-	x	x	x	x	-
Constructive Roofs from Solid Building Primitives	-	-	x	-	x	-	-
Procedural modeling of Architecture with Round Geometry	-	-	x	x	o	-	-
Curvature-controlled Curve Editing using Piecewise Clothoid Curves	-	-	-	-	-	-	x

Table 8.3.: Objects that the techniques can generate

8.3. Coded Representation

Figure 8.2 now shows a comparison of code snippets from different papers. The techniques that use Production Rule Systems (Figures 8.2a-8.2f) all have fairly similar syntax.

Most of them use coded rules where the label is written at the left side, then some kind of arrow pointing rightwards, followed by the specification of the rule on the right side. *Procedural Modeling of Cities* is the only L-System here, and the form of their rules is syntactically slightly different, but still uses the same principle of a Production Rule Systems.

The technique *Procedural modeling of Architecture with Round Geometry* (Figure 8.2h) presented in this thesis is also a Production Rule System. In contrast to the other techniques it was intended to be completely implemented in the syntax of a host language and therefore use structures and keywords of the host language. Nevertheless the principle is the same: a rule is encoded by first stating its label, and then the specification of the programming steps that the host language environments executes when the rule is activated.

Finally, *Component Based Modeling of Complete Buildings* (Figure 8.2g) is the only technique that is not a Production Rule System. It is structured like a general scripting language. The principle of how it works is based on queries that query an existing base of objects, retrieve the wanted objects, and then refine the objects and create new ones.

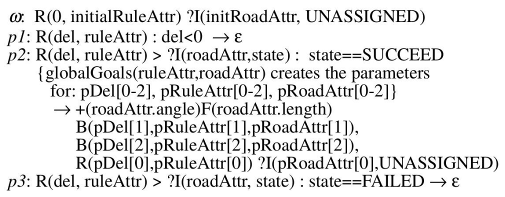
8.4. Insights

8.4.1. Creating a Domain Specific Method for a Modeling Domain

After the presentation of all the procedural modeling techniques one question arises:

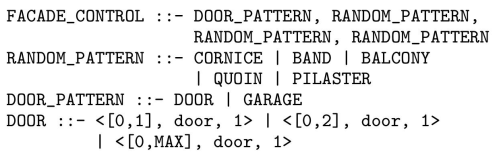
How can we take domains of 3D models and create domain specific methods for them?

The goal would be to give some kind of recipe for how to create them. Now, there are many possible design and implementation choices. It is very dependent on



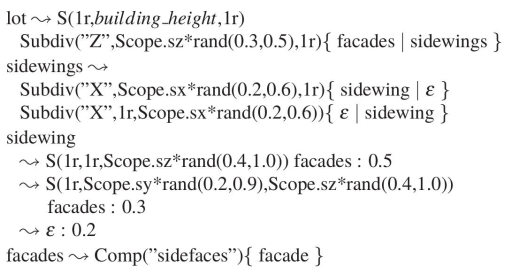
```
ω: R(0, initialRuleAttr) ?I(initRoadAttr, UNASSIGNED)
p1: R(del, ruleAttr) : del<0  → ε
p2: R(del, ruleAttr) > ?I(roadAttr,state) :  state==SUCCEED
    {globalGoals(ruleAttr,roadAttr) creates the parameters
      for: pDel[0-2], pRuleAttr[0-2], pRoadAttr[0-2]}
      → +(roadAttr.angle)F(roadAttr.length)
         B(pDel[1],pRuleAttr[1],pRoadAttr[1]),
         B(pDel[2],pRuleAttr[2],pRoadAttr[2]),
         R(pDel[0],pRuleAttr[0]) ?I(pRoadAttr[0],UNASSIGNED)
p3: R(del, ruleAttr) > ?I(roadAttr, state) : state==FAILED → ε
```

(a) Procedural Modeling of Cities



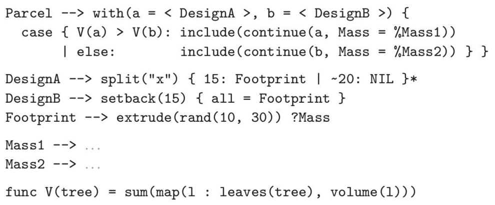
```
FACADE_CONTROL ::- DOOR_PATTERN, RANDOM_PATTERN,
                   RANDOM_PATTERN, RANDOM_PATTERN
RANDOM_PATTERN ::- CORNICE | BAND | BALCONY
                   | QUOIN | PILASTER
DOOR_PATTERN ::- DOOR | GARAGE
DOOR ::- <[0,1], door, 1> | <[0,2], door, 1>
         | <[0,MAX], door, 1>
```

(b) Instant Architecture



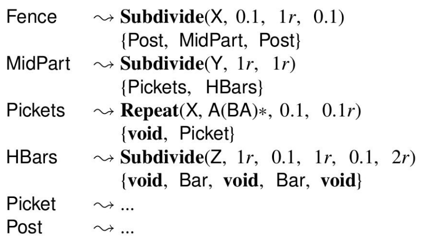
```
lot ⤳ S(1r,building_height,1r)
  Subdiv("Z",Scope.sz*rand(0.3,0.5),1r){ facades | sidewings }
sidewings ⤳
  Subdiv("X",Scope.sx*rand(0.2,0.6),1r){ sidewing | ε }
  Subdiv("X",1r,Scope.sx*rand(0.2,0.6)){ ε | sidewing }
sidewing
  ⤳ S(1r,1r,Scope.sz*rand(0.4,1.0)) facades : 0.5
  ⤳ S(1r,Scope.sy*rand(0.2,0.9),Scope.sz*rand(0.4,1.0))
      facades : 0.3
  ⤳ ε : 0.2
facades ⤳ Comp("sidefaces"){ facade }
```

(c) Procedural Modeling of Buildings



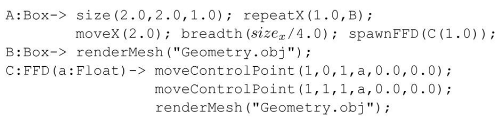
```
Parcel --> with(a = < DesignA >, b = < DesignB >) {
  case { V(a) > V(b): include(continue(a, Mass = %Mass1))
       | else:        include(continue(b, Mass = %Mass2)) } }

DesignA --> split("x") { 15: Footprint | ~20: NIL }*
DesignB --> setback(15) { all = Footprint }
Footprint --> extrude(rand(10, 30)) ?Mass

Mass1 --> ...
Mass2 --> ...

func V(tree) = sum(map(l : leaves(tree), volume(l)))
```

(d) Advanced Procedural Modeling of Architecture



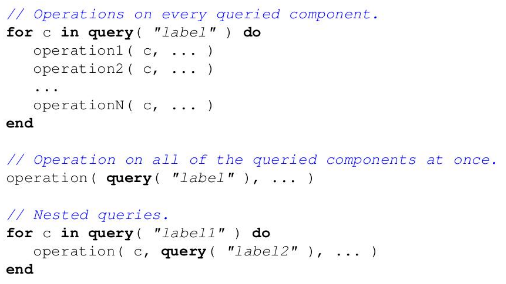
```
Fence     ⤳ Subdivide(X, 0.1, 1r, 0.1)
              {Post, MidPart, Post}
MidPart   ⤳ Subdivide(Y, 1r, 1r)
              {Pickets, HBars}
Pickets   ⤳ Repeat(X, A(BA)*, 0.1, 0.1r)
              {void, Picket}
HBars     ⤳ Subdivide(Z, 1r, 0.1, 1r, 0.1, 2r)
              {void, Bar, void, Bar, void}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

(e) Shape Grammars on Convex Polyhedra



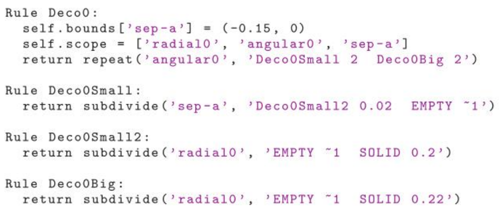
```
A:Box-> size(2.0,2.0,1.0); repeatX(1.0,B);
        moveX(2.0); breadth(size_x/4.0); spawnFFD(C(1.0));
B:Box-> renderMesh("Geometry.obj");
C:FFD(a:Float)-> moveControlPoint(1,0,1,a,0.0,0.0);
                 moveControlPoint(1,1,1,a,0.0,0.0);
                 renderMesh("Geometry.obj");
```

(f) Generalized Use of Non-Terminal Symbols for Procedural Modeling



```
// Operations on every queried component.
for c in query( "label" ) do
   operation1( c, ... )
   operation2( c, ... )
   ...
   operationN( c, ... )
end

// Operation on all of the queried components at once.
operation( query( "label" ), ... )

// Nested queries.
for c in query( "label1" ) do
   operation( c, query( "label2" ), ... )
end
```

(g) Component Based Modeling of Complete Buildings



```
Rule Deco0:
  self.bounds['sep-a'] = (-0.15, 0)
  self.scope = ['radial0', 'angular0', 'sep-a']
  return repeat('angular0', 'Deco0Small 2  Deco0Big 2')

Rule Deco0Small:
  return subdivide('sep-a', 'Deco0Small2 0.02  EMPTY ~1')

Rule Deco0Small2:
  return subdivide('radial0', 'EMPTY ~1  SOLID 0.2')

Rule Deco0Big:
  return subdivide('radial0', 'EMPTY ~1  SOLID 0.22')
```

(h) Procedural modeling of Architecture with Round Geometry

Figure 8.2.: Code snippets of some of the presented techniques (images taken from the respective papers).

the specific domain and problem and hard to generalize. It is also to some degree a matter of taste, but an attempt for an objective assessment is made here.

Simplicity

There is a famous quote made by Einstein that is very fitting for the design of many computer systems, and overall systems in general:

"Everything should be made as simple as possible, but no simpler."

One could also argue that in this context it could be rephrased as: "Everything should be made as simple as possible, and only as complex as necessary".

Especially for DSLs things should be as simple as possible, because a DSL should by definition only cover the domain (and not more) while also being very user friendly. Both requirements call for restrictiveness and simplicity.

For example, in the programming language family of Lisp, all code is expressions in lists. How this expressions are interpreted makes the difference. They can either be treated as pure data, functions, or macros.

- **Data**: Data is just information and does not do anything in itself, it has to be used by other code.
- **Functions**: Functions are executed. They take data and perform computations that produce new data.
- **Macros**: Macros at last, take data and functions and produce new functions (which in turn are executed later).

One can see that macros are clearly more powerful than functions, because every function can be implemented as a macro (which simply returns the function itself) but not vice-versa. And similarly, functions are clearly more powerful than data, because all data can be implemented as a function (which simply returns the data itself) but not vice-versa.

So while macros are more powerful than functions and these are more powerful than data, when one thinks of simplicity, it is the other way around.

Since data is not doing anything and is simply that what is written down (encoded) it is very simple and easy to understand. Functions on the contrary are much harder to understand because the whole computation that is performed by the functions must be understood. Finally, macros are even harder to understand, since they compute something (a function) that in turn itself computes something again. This level of indirection is the reason why macros are hard to understand for many programmers.

This is the reason why in Lisp it can be argued that code should be written in a way, such that it is whenever possible simple data, and only if needed a function, and only in very rare cases a macro. This idea attempts to make the code as simple and readable as possible.

Language concepts of domain specific methods

In the context of DSL design, similar ideas could be used in order to strive for simplicity. There are many different language concepts that a domain specific method could use.

The most simple are language concepts that are built upon pure data. These can be for example:

- **Tree data structures**: Data contained in a hierarchical tree. Found in formats such as JSON, XML, or YAML.
- **Records**: As in data entries in relational databases.
- **RDF triple**: Encoded information in Resource Description Frameworks.

More complex language concepts that have similarities to functions in programming languages are for example:

- **Production rule systems**: Processing of entities according to rules. Rule invocation is somewhat similar to function invocation in programming languages. Further described in Section 4.1.
- **Decision tables**: Similar rules to Production rule systems but arranged in a table. Further described in Section 4.1.

Even more complex language concepts with similarities to complete programming languages are:

- **State machines**: State machines describe program flow by transitions between abstract states. Further described in Section 4.1.

Some other, harder to categorize language concepts are:

- **Spreadsheets**: Tables that let users entry data or formulas known from programs such as Microsoft Excel.
- **Dependency networks**: Describe dependencies between entities. Further described in Section 4.1.

In choosing between these options, one of course has to evaluate which method suits the problem domain best, but can also strive for an as simple as possible solution.

Domain specific methods in the case studies

Arguably the simplest of the mentioned language concepts is the tree data structure. It consist of pure data that is hierarchically structured. Tree data structures can be encoded in many widely used formats (such as JSON, XML, or YAML), but also internally in a programming language (for example as nested dictionaries and lists in Python), while there is no need for complicated custom parsing. They can contain a lot of data in big structures. This can also lead to a disadvantage: If the amount of data gets too big, they can become confusing.

The tree data structure was chosen for the Case Study *Constructive Roofs from Solid Building Primitives* (See chapter 5). Different building blocks in the paper correspond to entries in the tree. Additional attributes of the building blocks are saved as sub-entries.

For the Case Study *Procedural Modeling of Architecture with Round Geometry* the production rule system was chosen. This system is the basis of shape grammars. A production rule system is still arguably simple in the sense that it behaves similar to a function - it takes input (a shape) and produces output (new shapes). When the production rule system is executed, new shapes are being created, and the information in the system is saved in attributes of those shapes.

8.4.2. A common Language

In computer science in general, the question of how a particular problem should be represented in code is often discussed.

A simple and straightforward approach is to encode the problem in some sort of data notation format. This can for example be XML, JSON, or YAML. The advantage is that this encoding is very simple and does not need operators or complicated logic. This can also make it less error-prone. For things were there is a lot of repetition a disadvantage is that everything needs to be specified and cannot be generated via operators. Of course, many complicated facts cannot even be stated, since the needed operators or logic is not present.

Another problem can be that there might be a multitude of possible ways to specify the same thing. For example, in order to define an axis aligned box one could specify one point of the box together with its width, depth and height, or one point together with the opposite point of the box. If boxes are now specified in different ways, all those possibilities need to be implemented in the software that reads the specifying files and needs to be understood by the users.

Encoding a problem in a common programming language instead of a data notation format provides much more flexibility. Instead of having different data specifications, operators can generate the needed primitives and they can be saved internally in a unique way (for example, boxes could internally always be saved by a point plus width, depth, and height). Having operators also provides the opportunity to do arbitrary calculations in order to generate the primitives. The disadvantage of ordinary programming is of course that it is much more complex than a simple data notation format and also more error-prone.

Analogue to the general class of computer science problems, shapes can also be represented in different ways. There are in general two ways how this is done.

One way that shapes can be represented is by primitives that make up a particular shape. These can be curves such as spline or clothoid curves, surfaces such as triangle meshes, subdivision surfaces, or NURBS surfaces, or solids defined by a boundary representation or convex polyhedra. Either way, the description defines the shape, but not how it is constructed.

The other way that shapes can be represented is by construction. The execution of the construction steps recreates the shape. How, and in which order the construction steps are executed is the topic for procedural modeling. Procedural models are basically programs that are run, which then execute a series of construction steps usually performed by API calls to a low-level geometric primitive library.

While the first representation is usually simpler, less error prone, and immediately examinable, the second representation has other advantages: it is usually easier to modify, or parameterize. When one chooses the procedural representation, the question then becomes: What is an ideal way to represent the model procedurally?

One interesting fact from computer science is that once a programming language is touring complete it can basically perform all possible tasks, just as any other touring complete language. Since most of modern non-trivial languages are touring complete it concludes that they can all generate the same shapes. The important aspect now becomes: What language has the suited expressive power for the problem domain?

The Generative Modeling Language

The Generative Modeling Language (GML) is described in [38] and [39] and is a language that was made with the primary focus on 3D modeling. It is derived from postscript [71] and is therefore a simple stack-based language. Nevertheless, it is a fully touring complete language as most modern programming languages. Its simplicity and lack of almost any syntax might strike some as a surprise, but it is this simplicity that brings some certain advantages.

Because of its simplicity it is easy to write an interpreter for GML (as has been done at the Institute of Computer Graphics and Knowledge Visualization at the TUGraz). And on the other hand, it is also easy to generate GML code from other sources, such as higher level programming languages, domain specific languages, or graphical user interfaces. GML is also easy to use as a language where other systems are embedded in. This makes GML very suitable as a common intermediate language for 3D models that is not necessarily used directly to write procedural 3D models, but is generated or used by other systems.

Existing applications in GML

In the case of domain specific languages a particular interesting possibility is to embed the DSL as an internal DSL into GML. This, for example was done for the shape grammars in the papers [87, 98, 26], which are forms of production rule system DSLs.

There are also declarative specifications where objects are specified in GML data structures (such as lists and dictionaries) which are then evaluated. This is the case in [25] and [85].

For graphical user interfaces, GML can serve as the framework that handles graphical primitives and the visualization of 3D models. The papers [41] and [86] use this approach.

The Euclides framework [80] deals with the translation of JavaScript to GML (or any other PostScript dialect) because JavaScript is a much more known language than GML. For this approach, for one part, JavaScript expressions have to be translated. This is relatively straight-forward since it is mostly a simple infix to postfix conversion. For the other part, all the control flow structures also have to be translated. This is non-trivial since PostScript has no 'goto' operation and all the control flow structures of JavaScript are translated into the respective PostScript control flow structures.

Interoperability through a common language

The main advantage of having a language like GML as a common language for procedural 3D modeling is interoperability of different procedural techniques. Different techniques can use completely different paradigms, but when they use the same base language or encoding (such as GML) their results can be combined. For example, this was the case in the paper [25], where the roofs and building shells were modeled with the technique described in the paper, but the facades were modeled with a classical split grammar approach as described in [87]. Another possibility could be to model building walls along curves (e.g. via splines or clothoids as described in [41]) and then use a classical split grammar again for the facades.

In this sense GML has similarities to XML or bytecode. XML provides a common encoding for many specification formats (XML dialects) and abolishes the need to have an individual parser for every one. Bytecode serves as a common ground for different programming languages that can be interpreted by the same virtual machine. GML in analogy can serve the role of a unifying basis were other systems, GML dialects, or domain specific languages can built upon.

GML today already has a multitude of primitives for 3D modeling included. Notably it can represent surfaces with combined b-reps. These can either be standard triangle meshes, subdivision surfaces, or a combination of both. Combining both variants in one representation allows shapes to have smooth surfaces as well as sharp creases and corners. This was used in previous work where gothic windows [40] or a pipe systems for a city [61] was created. The other representation are convex polyhedra, implemented with special precision arithmetic. They provide a solid form of representation and allow constructive solid geometry (CSG) operations. This means that the union, intersection, or difference of shapes can be computed. This was used for the implementation of the shape grammars [87, 26], the windows in [85], and the roofs in [25].

9. Conclusion

This thesis was concerned with the problem of modeling historic buildings in the computer in a way that can avoid immense manual modeling effort for the ever growing demands in computer graphics. For this, procedural modeling is the method of choice, since it can generate geometry automatically to some degree.

While existing research in procedural modeling can produce satisfying results for box-shaped buildings, more complex forms are harder to generate. The forms of complex historical buildings can be grouped into different classes. In order to model these classes, or domains, different domain specific approaches can be sought after.

9.1. Contribution

The full scope of modeling domains necessary for historic buildings is too big for one thesis, but three distinct problem domains were examined. For this domains new domain specific modeling techniques were developed and described here. These techniques comprise the modeling of complex roof landscapes for historic buildings and historic cities, the modeling of round parts and details of historic buildings, and the modeling of ornamental forms in historic buildings.

Roofs The technique about modeling roof landscapes uses building blocks representing individual parts of a building. These are combined to form a complete building and an automatic trimming algorithm is used in order to cut off surplus geometry. The building blocks can also generate the basic wall geometry of the building that can be refined with other methods (e.g. conventional procedural modeling techniques) to create a complete building.

Round Building Geometry The technique concerned with round building geometry represents each round part of a building in its own coordinate system. These coordinate systems can be for example cylindrical, spherical, or conical. The choice of coordinate system naturally leads to round geometry and allows to use existing procedural modeling techniques in the respective coordinate space. This allows the procedural generation of towers, domes, or arches of historical buildings.

Ornamental Forms The technique using piecewise clothoid curves shows the further development of a special curve representation - the clothoid - that has very favorable curvature properties, and can capture the forms of ornamental elements with only few control points. Compared to the widely used spline curve representation some advantages of using a piecewise colothoid curve are that the curve passes through its control points that can be directly modified, the invariance of the curve in respect to inserting control points, that control points are only needed where deviation from linear curvature is needed, and that the curve can be constrained by tangent or curvature values.

Besides developing these techniques some general insights to creating domain specific approach for specific modeling problems were gained. Domain specific languages are a long studied field in computer science and some lessons can be learned there. There is always a multiplicity of options for creating a domain specific technique for a particular problem domain. Principles of software design such as keeping a system as simple as possible to avoid accidental complexity, also apply here.

Further the benefits of having a common language for procedural modeling were highlighted. All models in this thesis were created with the help of a certain procedural modeling language - the GML. Using one common base language allows the interoperability of different procedural techniques.

9.2. Benefit

The works presented in this thesis introduce benefits to a wide class of applications that make extensive use of 3D models. The following fields can all benefit from using procedural modeling techniques. However, as laid out already, when it comes to the generation of historic buildings they are limited. With the new techniques introduced in this thesis, historic buildings can now be generated in higher detail and accuracy, which advances the creation 3D models of historic buildings, towns, and cities.

Virtual Worlds As already mentioned, virtual worlds nowadays can include a vast amount of 3D models, including countless building models. To create these virtual worlds, often an army of artists has to work for years to create all the required models. One obvious solution to lessen the workload is automatic procedural generation of 3D models. Especially building models with their regularities and repetitions are very well suited for procedural generation. Today, virtual worlds are heavily used in movies and video games, and the upcoming virtual reality (VR) technology will only increase the demand. Virtual worlds with historic cities will benefit from the presented new techniques.

3D Scanning and Reconstruction 3D laser scanning is a technique that can scan an objects, which yields to a resulting dense point cloud and often a following dense triangle mesh. The detail of the result is usually high, but also is the amount of data that is generated. The resulting geometry is also hardly modifiable. To produce a scalable result that can also be further adapted, inverse procedural modeling techniques are used. These techniques use a given procedural 3D model and adapt its parameters in order to closely fit a 3D scan. When historic buildings are scanned, the newly presented techniques will increase the quality of the resulting models.

Cultural Heritage In order to preserve the human cultural heritage many initiatives are starting to scan important cultural sights with laser scanning. Here, the quality and usability of the scanned result can also be increased with the mentioned inverse procedural modeling techniques. The reconstruction of such cultural sights can benefit greatly from the new techniques, since exiting ones can reproduce them only in limited fashion.

Urban Planning Civil engineering already uses virtual 3D models of cities for urban planning. One example is the company ESRI [28] whose City Engine [21] software is used to a great degree for urban planning. Another example is the Virtual Singapore [93] project. A project that aspires to create a complete 3D model of Singapore. While existing procedural techniques are suitable to model the building shells of most buildings in modern cities such as Singapore, the new techniques will help reproducing historic parts of European cities in greater detail.

Mapping Applications Mapping applications include 3D models of many cities that are mostly generated by 3D scanning and reconstruction techniques. The initial result usually has far too much data and needs to be simplified. The resulting geometry is often not clean, meaning walls, windows, roofs, etc. are slightly distorted and not one onehundred percent planar. Inverse procedural modeling can solve this problem. As for urban planning, the presented new techniques will aid in the creation of mapping models of historic cities.

9.3. Validation of Research Hypothesis

In Section 3.3 a list of research hypotheses was given. After presentation and evaluation of the newly presented techniques in Chapters 5, 6, 7, and 8 these hypothesis are revisited here.

- **H1**: *Historic buildings cannot be reasonably modeled with existing procedural shape modeling techniques.*

  This is shown in Chapter 2 and especially in Figure 1.1 and Section 2.3. The existing techniques can model box-like buildings very well, but the modeling capabilities for more elaborate forms are limited.

- **H2**: *Domain Specific Languages (DSLs) are well suited for the extension of procedural shape modeling techniques to model historic buildings.*

  Chapter 4 gives an introduction to Domain Specific Languages. Domain Specific Methods are then used in the later Case Study Chapters. Especially, the for architectural modeling widely used split grammar approach, can be further adapted to work with different methods as shown in Chapter 6. Guidelines for the creation of Domain Specific Methods can be found in Section 8.4.1.

- **H3**: *The three most effective extensions for historical buildings are: Roof landscapes, round building geometry, and free form curves.*

  The Case Study Chapters 5, 6, and 7 cover all of these potential extensions and give multiple results and applications.

- **H4**: *The DSLs for the extensions can all be formulated using a common underlying formalism.*

  Section 8.4.2 explains the arguments for a common base language in which different Domain Specific Languages are implemented.

9.4. Publications

The novel works presented in this thesis have been published and peer reviewed in following international conferences and journals:

The Case Study *Constructive Roofs from Solid Building Primitives* from Chapter 5 was:

- presented at the *2014 International Conference on Cyberworlds* in Santander, Spain.
- published in the conference proceedings of *Cyberworlds (CW), 2014 International Conference on. IEEE. 2014, pp. 63–70* [24].
- published as extended journal version in *Transactions on Computational Science XXVI. Springer, 2016, pp. 17–40* [25].

The Case Study *Procedural Modeling of Architecture with Round Geometry* from Chapter 6 was:

- presented at the *2016 International Conference on Cyberworlds* in Chongqing, China.
- published in the conference proceedings of *Cyberworlds (CW), 2016 International Conference on. IEEE. 2016, pp. 81–88* [27].
- published as extended journal version in *Computers & Graphics 64 (2017), pp. 14–25* [26].

The Case Study *Curvature-controlled Curve Editing using Piecewise Clothoid Curves* from Chapter 7 was:

- presented at the *2013 Shape Modeling International (SMI)* conference in Bournemouth, United Kingdom.
- published in *Computers & Graphics 37.6 (2013), pp. 764–773* [41].

10. Future Work

10.1. Procedural Modeling of Interiors of Buildings

Currently split grammars are mostly used for the mass generation of building shells for whole cities, whereas detailed modeling of interiors of buildings is partly neglected. The principles of split grammars are foremost designed to model facades of buildings.

Interiors are harder to realize, since they need to respect all the dependencies inside the building. Interiors have been done with other procedural techniques, but these are focused on simpler residential buildings and are not able to reproduce interior geometry as found for example in complex historical buildings.

There are several drawbacks to the current split grammar systems that limit their range of useful applications for interiors. Some drawbacks are (see Table 10.1):

1. Since simple split grammar systems use boxes as their primitives, everything that is not rectangular cannot be modeled using concise grammar rules based on splits (see example 1 in Table 10.1). Instead, the user is forced to employ a general scripting language in order to model non-rectangular geometry.

2. Alignment of different structures is a common requirement when modeling architecture (see example 2 in Table 10.1). To align multiple elements along a common axis requires additional rules to be added to the existing grammar. This is particularly labor intensive and error prone.

3. Parts of an object can be arranged in such a way, that in order to model them there is no way of splitting them apart without splitting through the interior of one part (see example 3 in Table 10.1). In this case the two halves have to be modeled separately and have to be manually aligned (as covered in point 2.).

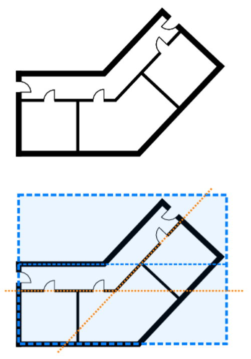
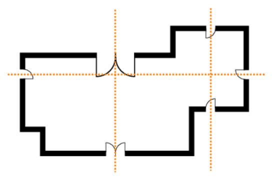
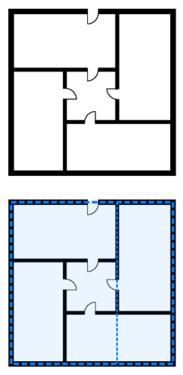
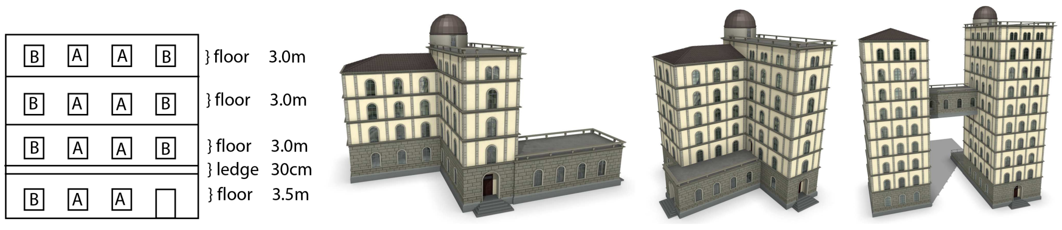
		
1. Non-rectangular geometry	*2. Alignment of geometry*	*3. Unsplittable geometry*
When geometry is non-rectangular, a split along the main axis of the rectangular bounding box (blue) cannot partition the geometry in a useful way. In this example, the floor plan follows two axis (orange) that are not perpendicular to each other.	When geometry elements (here doors and windows) are aligned along axes (orange), it can be hard to reproduce the positions with split grammars because alignment information is lost during splits.	Elements of the geometry can be arranged such that a split (blue) is not possible without splitting right through one element. This situation is even more likely when there is non-convex geometry involved.

Table 10.1.: Limitations of split grammars that complicate the modeling of building interiors.

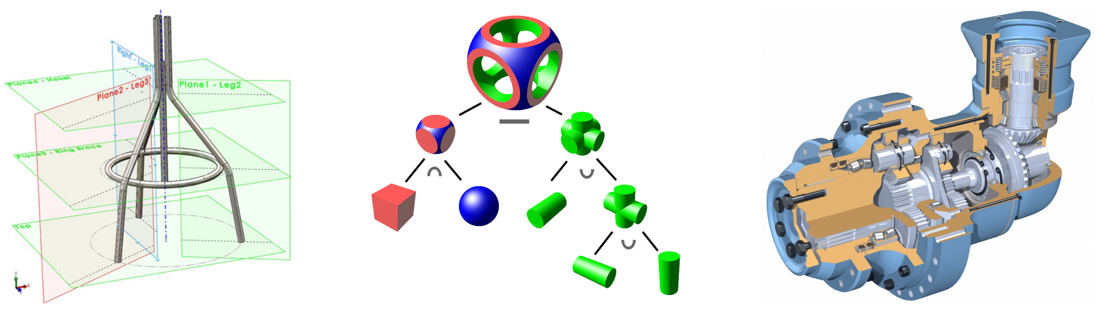
10.2. Highly detailed Procedural Modeling

The efficiency of a modeling method depends on how rich in detail the final model should be. Even the most complex object can be modeled with a trivial method if the final model can be an approximation of the object with low detail. But when more detail and more geometric properties have to be included in the model, current procedural methods reach their limits.

For this, an approach would be needed where the focus lies on high detail modeling and not on randomized mass generation. As long as a model has details that follow certain rules, procedural approaches can bring much reuse in the modeling process and therefore save work.



(a) Split grammars use a rule system to describe 3D models. The model is automatically generated by successively splitting space into smaller blocks according to the rules. Here a building is generated with a split grammar. Various instances are easily obtained through reparametrization. Images taken from [65].



(b) CAD uses various techniques like construction planes and boolean CSG operators for 3D modeling. With CAD, high detail models can be manually created for different fields like architecture and engineering. Images taken from [6, 7, 8].

Figure 10.1.: Split grammar models (a) and CAD models (b) are obtained using different techniques for model creation.

10.3. Extension of Procedural Modeling to Construction Parts

Another direction where procedural modeling could expand to, and where detailed modeling is required, is the modeling of construction parts. These parts are usually modeled with special modeling software tailored to Computer-aided design (CAD).

In CAD the focus is not mass modeling but the exact construction of geometry for industrial production. Here, to clip and constrain geometry, construction lines and planes are used (See Figure 10.1). A similar construction method incorporated in a split grammar approach may hold the possibility to extend procedural methods to this field.

Appendix

Appendix A.

Code for Chapter 5

```
$S$ = arbitrary structure
for $A \in$ solids of $S$ do
    for $a \in$ sides of $A$ do
        $L_{solid}$ = empty list
        for $B \in$ solids of $S$ where $A \neq B$ do
            $L_{side}$ = empty list
            for $b \in$ sides of $B$ do
                if $b$ rises above $a$ then
                    $g$ = inner side-geometry of $b$
                else
                    $g$ = outer side-geometry of $b$
                end
                append $g$ to $L_{side}$
            end
            $I = \cap$ of all in $L_{side}$
            append $I$ to $L_{solid}$
        end
        $T$ = outer sub-solid of $A - (\cup$ of all in $L_{solid})$
        render the roof face (according to $a$) of $T$
    end
end
```

Algorithm 1: Extracting the geometry for the roof of one structure. For simplicity, the algorithm shown here only covers sides with one roof element and not with a profile-polygon (as in Figure 5.12).

Table A.1.: Description of Algorithm 1.

Line	Description
1	In our system different structures have independent roofs so we can calculate the roof independently for one structure.
2,3	We calculate the roof face for each side of a solid separately.
4	The roof of the solid changes when another solid intersects with it. We will trim the solid with all other solids, but because of possible double-covered areas we have to take either the inner or outer side-geometry of another solid, depending on the situation. Therefore we create a new list where we will save the new other solids with inner/outer side-geometry.
5	We iterate over all other solids of the house.
6-13	For every side of the other solid we decide whether it rises over the side we are currently calculating (and therefore forms a double-covered area). The test for rising over the other side is described in Section 5.5.1. If yes, we take the inner side-geometry. If no, we take the outer side-geometry. We collect the side-geometries in a new list.
15-16	We build the boolean intersection of all the elements in the list and thereby form a custom sub-solid that respects the double-covered areas for each of its sides.
18	All the custom sub-solids are now boolean subtracted from the outer sub-solid of our initial solid. This removes all areas from the roof where another solid intersects.
19	The new roof of this side can now be rendered.

Appendix B.

Code for Chapter 6

attribute	datatype	default value	description
coordsystems	dict<(coord-system, name, name, name)>		Coordinate systems. The keys are the coordinate system names, coord-system is a specification, and the three names are the names of the coordinates.
scope	list<name>		List of coordinate names whose bounds form the scope of the shape.
bounds	dict<(number, number)>		Dimension (start and end value) along a coordinate, keys are coordinate names.
parent	shape	the parent shape	Reference to the parent shape (automatically set).
user	dict		User attributes (arbitrary).
children	dict<list<shape>>		Child shapes. The user can choose the keys arbitrarily (e.g. ”basement”, ”floor”, ”roof”, etc.).
geometry	list<geometry>	union of all child geometries	Final geometry - will be passed to the parent.

Table B.1.: Description of the main attributes in our implementation. Split rules save the resulting sub-shapes in the children attribute (under a given name). Per default, the geometry attribute results in the union of all child geometries. This, however, can be overridden by the user and gives the possibility to perform Boolean operations.

```
Rule Start:
  # Setup of a cylindrical coordinate system. The first vector is the origin, the
      others are for orientation:
  coord = create-coordinate-system('cylindrical', (0,0,0), (1,0,0), (0,1,0),
      (0,0,1))
  # The coordinate system is saved in the attribute 'coordsystems' along with the
      names of its coordinates:
  self.coordsystems['vault0']    = (coord, 'radial0', 'angular0', 'h0')
  # The bounds of all three coordinates are set
  self.bounds['radial0']         = (1.7, 2)
  self.bounds['angular0']        = (0, 360)
  self.bounds['h0']              = (-3, 3)
  # The same for the second cylindrical coordinate system:
  coord = create-coordinate-system('cylindrical', (0,0,0), (0,1,0), (1,0,0),
      (0,0,1))
  self.coordsystems['vault1']    = (coord, 'radial1', 'angular1', 'h1')
  self.bounds['radial1']         = (1.7, 2)
  self.bounds['angular1']        = (0, 360)
  self.bounds['h1']              = (-3, 3)
  # A Cartesian coordinate system is needed to cut out geometry, and for the edge
      decoration:
  coord = create-coordinate-system('cartesian', (0,0,0), (1,1,0), (-1,1,0), (0,0,1)
      )
  self.coordsystems['seperator'] = (coord, 'sep-a', 'sep-b', 'sep-c')
  # Calling of the rules for both vaults, and the edge decoration:
  self.children['Vault0']        = call-rule('Vault0')
  self.children['Vault1']        = call-rule('Vault1')
  self.children['Decoration']    = call-rule('Decoration')
  # The final geometry automatically results to the union of all geometry of
      elements in the 'children' attribute.

Rule Vault0:
  # The scope is set to be bound by the coordinates from the 'vault0' coordinate
      system:
  self.scope = ['radial0', 'angular0', 'h0']
  # The vault is split along the coordinate system axis into thin and thick pieces,
       with bounds 0.1 and 0.7:
  self.children['Main'] = repeat('h0', 'Vault0Thin 0.1  Vault0Thick 0.7')
  # Cutout geometry is created in order to cut away geometry from this vault where
      the other vault will intersect:
  self.children['Cutout0'] = call-rule('Cutout0')
  self.children['Cutout1'] = call-rule('Cutout1')
  # The final geometry is manually set as the main geometry minus the cutout
      geometries:
  self.geometry = difference('Main', union('Cutout0', 'Cutout1')

Rule Vault0Thin:
  # The inner part of this slice becomes empty, and the outer part filled with
      solid geometry
  # (bound by the scope - the bounds in 'radial0', 'angular0', 'h0').
  # The '~1' term specifies that this value is relative, and adjusts to the space
      that is left:
  return subdivide('radial0', 'EMPTY ~1  SOLID 0.2')
  # Because there is a return value, its geometry will automatically be the final
      geometry of this shape.

Rule Vault0Thick:
  # The bigger parts are further split angular into the horizontal bars, and the
```

```
      inner cassettes:
  return repeat('angular0', 'Vault0Horizontal 5  Vault0Inner 20')

Rule Vault0Horizontal:
  return subdivide('radial0', 'EMPTY ~1  SOLID 0.17')

Rule Vault0Inner:
  return subdivide('radial0', 'EMPTY ~1  SOLID 0.1')

Rule Cutout0:
  # The bounds are manually set to form a wedge, that is later (in Rule Vault0) cut
       out on one side of the vault:
  self.dimension['sep-a'] = (0, +inf)
  self.dimension['sep-b'] = (-inf, 0)
  self.scope = ['sep-a', 'sep-b']
  return call-rule('SOLID')

Rule Decoration:
  # The decoration consits of 8 parts, the intersection of the vaults forms 2 edges
       that intersect,
  # giving 4 edge segments, and we model the left and the right side for each edge
      segment:
  self.children['Deco0'] = call-rule('Deco0')
  ...
  self.children['Deco8'] = call-rule('Deco8')

Rule Deco0:
  self.bounds['sep-a'] = (-0.15, 0)
  self.scope = ['radial0', 'angular0', 'sep-a']
  return repeat('angular0', 'Deco0Small 2  Deco0Big 2')

Rule Deco0Small:
  return subdivide('sep-a', 'Deco0Small2 0.02  EMPTY ~1')

Rule Deco0Small2:
  return subdivide('radial0', 'EMPTY ~1  SOLID 0.2')

Rule Deco0Big:
  return subdivide('radial0', 'EMPTY ~1  SOLID 0.22')
```

Listing B.1: Pseudo code (not the actual code) for the model in Figure 6.5 in a python-like syntax. 'self' refers here to the dictionary containing the attributes shown in Figure B.1. The word 'Rule' marks the beginning of a rule function. Not all rule functions are shown (e.g. 'Vault1', 'Cutout1', 'Deco1', etc.). However, these are coded in a similar way. Coding all of them would lead to a vast amount of code. In practice, we support parameters for our rule functions and can therefore reuse them. This reduces the number of them which is needed. We do not shown this here, in order to reduce complexity.

Bibliography

[1] URL: http://luxurydesign.vn/resources/upload/60177686.jpg (cit. on p. 31).

[2] URL: http://dhwcor.xyz/roman-vault-architecture/structure-palace-arch-decoration-free-roman-vault-architecture-images-building-the-aisle.html (cit. on p. 31).

[3] URL: https://www.albertmilde.com/img/sturany7.jpg (cit. on p. 32).

[4] URL: https://www.cypherpunk.at/files/2015/12/bspline_0.jpg (cit. on p. 33).

[5] URL: https://en.wikipedia.org/wiki/Euler_spiral (cit. on p. 33).

[6] URL: http://www.thefabricator.com/article/shopmanagement/-shop-technology-and-3-d-cad-techniques-for-modeling-bent-and-welded-tubing (cit. on p. 147).

[7] URL: http://en.wikipedia.org/wiki/Constructive_solid_geometry (cit. on p. 147).

[8] URL: http://www.danube-engineering.eu/kontakt.html (cit. on p. 147).

[9] Oswin Aichholzer et al. *A novel type of skeleton for polygons*. Springer, 1996 (cit. on p. 48).

[10] Jamaludin Md Ali et al. "The generalised Cornu spiral and its application to span generation." In: *Journal of Computational and Applied Mathematics* 102.1 (1999), pp. 37–47 (cit. on p. 103).

[11] Babak Ameri and Dieter Fritsch. "Automatic 3D building reconstruction using plane-roof structures." In: *ASPRS, Washington DC* (2000) (cit. on p. 47).

[12] U.H. Augsdoerfer, N.A. Dodgson, and M.A. Sabin. "Artifact analysis on B-splines, box-splines and other surfaces defined by quadrilateral polyhedra." In: *Computer Aided Geometric Design* 28.3 (2011), pp. 177–197. ISSN: 0167-8396 (cit. on p. 117).

[13] Franz Aurenhammer. "Weighted skeletons and fixed-share decomposition." In: *Computational Geometry* 40.2 (2008), pp. 93–101 (cit. on p. 49).

[14] Ilya Baran, Jaakko Lehtinen, and Jovan Popovic. "Sketching Clothoid Splines Using Shortest Paths." In: *Comput. Graph. Forum* 29.2 (2010), pp. 655–664 (cit. on pp. 103, 122).

[15] Alexander G. Belyaev. "A Note on Invariant Three-Point Curvature Approximations (Singularity theory and Differential equations)." In: *RIMS Kokyuroku* 1111 (Aug. 1999), pp. 157–164. ISSN: 1880-2818 (cit. on p. 106).

[16] David Ben-Haim, Gur Harary, and Ayellet Tal. "Piecewise 3D Euler spirals." In: *Proceedings of the 14th ACM Symposium on Solid and Physical Modeling*. SPM '10. Haifa, Israel: ACM, 2010, pp. 201–206. ISBN: 978-1-60558-984-8 (cit. on pp. 104, 124).

[17] Saffron Blaze. *Wikimedia commons - Leaning tower or Pisa*. License: CC BY-SA 3.0/Saffron Blaze. URL: https://upload.wikimedia.org/wikipedia/commons/6/66/The_Leaning_Tower_of_Pisa_SB.jpeg (cit. on pp. 3, 126).

[18] *Blender - a 3D modelling and rendering package*. Blender Institute, Amsterdam, 2016. URL: http://www.blender.org (cit. on p. 92).

[19] Martin Bokeloh, Michael Wand, and Hans-Peter Seidel. "A connection between partial symmetry and inverse procedural modeling." In: *ACM Transactions on Graphics (TOG)*. Vol. 29. 4. ACM. 2010, pp. 1–10 (cit. on p. 75).

[20] George Merrill Chaikin. "An algorithm for high-speed curve generation." In: *Computer Graphics and Image Processing* 3.4 (1974), pp. 346–349. ISSN: 0146-664X (cit. on p. 113).

[21] *City Engine*. URL: http://www.esri.com/software/cityengine (cit. on p. 141).

[22] Francois de Dijon/CC BY-SA 4.0. *Wikimedia commons - Gate of Honour in Versailles*. License: CC BY-SA 4.0/Francois de Dijon. URL: https://upload.wikimedia.org/wikipedia/commons/7/73/Ch%C3%A2teau_de_Versailles_-_grille_royale_01.jpg (cit. on pp. 3, 126).

[23] Dirk Dörschlag, Gerhard Gröger, and Lutz Plümer. "Semantically enhanced prototypes for building reconstruction." In: *Stilla, U. et al.(Eds.). Proc. of PIA'07, Intern. Archives of ISPRS* 36 (2007) (cit. on p. 48).

[24] Johannes Edelsbrunner et al. "Constructive roof geometry." In: *Cyberworlds (CW), 2014 International Conference on*. IEEE. 2014, pp. 63–70 (cit. on pp. 75, 143).

[25] Johannes Edelsbrunner et al. "Constructive roofs from solid building primitives." In: *Transactions on Computational Science XXVI*. Springer, 2016, pp. 17–40 (cit. on pp. 43, 75, 137, 138, 143).

[26] Johannes Edelsbrunner et al. "Procedural modeling of architecture with round geometry." In: *Computers & Graphics* 64 (2017), pp. 14–25 (cit. on pp. 71, 136, 138, 143).

[27] Johannes Edelsbrunner et al. "Procedural Modeling of Round Building Geometry." In: *Cyberworlds (CW), 2016 International Conference on*. IEEE. 2016, pp. 81–88 (cit. on pp. 75, 143).

[28] *ESRI*. URL: `https://www.esri.com/en-us/home` (cit. on p. 141).

[29] Dieter Finkenzeller. "Detailed building facades." In: *Computer Graphics and Applications, IEEE* 28.3 (2008), pp. 58–66 (cit. on p. 48).

[30] Dieter Finkenzeller. *Modellierung komplexer Gebäudefassaden in der Computergraphik*. KIT Scientific Publishing, 2008 (cit. on p. 48).

[31] Dieter Finkenzeller, Jan Bender, and Alfred Schmitt. "Feature-based decomposition of façades." In: *Proc. Virtual Concept*. December. 2005, pp. 1–9 (cit. on p. 80).

[32] Andre Fischer et al. "Extracting buildings from aerial images using hierarchical aggregation in 2D and 3D." In: *Computer Vision and Image Understanding* 72.2 (1998), pp. 185–203 (cit. on p. 48).

[33] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010 (cit. on pp. 35, 37, 38, 40, 42).

[34] *Freeimages*. 2016. URL: `http://www.freeimages.com/photographer/reneejvt-49071` (cit. on pp. 74, 96).

[35] *Freeimages*. 2016. URL: `http://www.freeimages.com/photographer/suemax-46273` (cit. on p. 74).

[36] Armin Grün. "Semi-automated approaches to site recording and modeling." In: *International Archives of Photogrammetry and Remote Sensing* 33.B5/1; PART 5 (2000), pp. 309–318 (cit. on p. 47).

[37] Xuekun Guo et al. "Creature grammar for creative modeling of 3D monsters." In: *Graphical Models* 76.5 (2014), pp. 376–389 (cit. on p. 75).

[38] Sven Havemann. "Generative mesh modeling." In: *PhD Thesis, Technische Universitaet Braunschweig, Germany* 1.1-303 (2005), pp. 4–4. DOI: www.digibib.tu-bs.de/?docid=00000008. URL: http://generalized-documents.org/CGVold/DigitalLibrary/publications/TechnicalReports/bs/TR-tubs-cg-2003-01.pdf (cit. on pp. 86, 91, 136).

[39] Sven Havemann. "Generative modellierung." PhD thesis. Citeseer, 1997 (cit. on p. 136).

[40] Sven Havemann and Dieter Fellner. "Generative parametric design of gothic window tracery." In: *Shape Modeling Applications, 2004. Proceedings*. IEEE. 2004, pp. 350–353 (cit. on p. 138).

[41] Sven Havemann et al. "Curvature-controlled curve editing using piecewise clothoid curves." In: *Computers & Graphics* 37.6 (2013), pp. 764–773 (cit. on pp. 99, 137, 144).

[42] Mark A. Heald. "Rational Approximations for the Fresnel Integrals." English. In: *Mathematics of Computation* 44.170 (1985), ISSN: 00255718 (cit. on pp. 101, 104).

[43] Hai Huang and Claus Brenner. "Rule-based roof plane detection and segmentation from laser point clouds." In: *Urban Remote Sensing Event (JURSE), 2011 Joint*. IEEE. 2011, pp. 293–296 (cit. on p. 47).

[44] Hai Huang, Claus Brenner, and Monika Sester. "3d building roof reconstruction from point clouds via generative models." In: *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM. 2011, pp. 16–24 (cit. on p. 48).

[45] Hai Huang, Claus Brenner, and Monika Sester. "A generative statistical approach to automatic 3D building roof reconstruction from laser scanning data." In: *ISPRS Journal of Photogrammetry and Remote Sensing* 79 (2013), pp. 29–43 (cit. on p. 48).

[46] *Institute of Computer Graphics and Knowledge Visualisation (CGV), Graz University of Technology*. URL: https://www.tugraz.at/institute/cgv/home/ (cit. on pp. 18, 20, 24, 43, 71, 99).

[47] Xiaogang Jin and VF Li. "Three-dimensional deformation using directional polar coordinates." In: *Journal of Graphics Tools* 5.2 (2000), pp. 15–24 (cit. on p. 75).

[48] Tom Kelly and Peter Wonka. "Interactive architectural modeling with procedural extrusions." In: *ACM Transactions on Graphics* 30.2 (2011), pp. 1–15. ISSN: 07300301. DOI: 10.1145/1944846.1944854. URL: http://portal.acm.org/citation.cfm?doid=1944846.1944854 (cit. on pp. 49, 80).

[49] KyoHyouk Kim and Jie Shan. "Building roof modeling from airborne laser scanning data based on level set approach." In: *ISPRS Journal of Photogrammetry and Remote Sensing* 66.4 (2011), pp. 484–497 (cit. on p. 47).

[50] Lars Krecklau and Leif Kobbelt. "Procedural modeling of interconnected structures." In: *Computer Graphics Forum*. Vol. 30. 2. Wiley Online Library. 2011, pp. 335–344 (cit. on p. 47).

[51] Lars Krecklau, Darko Pavic, and Leif Kobbelt. "Generalized use of non-terminal symbols for procedural modeling." In: *Computer Graphics Forum* 29.8 (2010), pp. 2291–2303. ISSN: 01677055. DOI: 10.1111/j.1467-8659.2010.01714.x (cit. on pp. 20, 21, 47, 75, 80).

[52] Robert G Laycock and AM Day. "Automatically generating roof models from building footprints." In: (2003) (cit. on p. 48).

[53] Luc Leblanc, Jocelyn Houle, and Pierre Poulin. "Component-based modeling of complete buildings." In: *Proceedings of Graphics Interface 2011* (2011), pp. 87–94. ISSN: 07135424. URL: http://dl.acm.org/citation.cfm?id=1992917.1992932 (cit. on pp. 21, 22, 79).

[54] Thomas Ledl. *Wikimedia commons - Karlskirche*. License: CC-BY-SA 4.0/Thomas Ledl. URL: https://upload.wikimedia.org/wikipedia/commons/3/39/Karlskirche_Abendsonne_1.jpg (cit. on pp. 3, 126).

[55] Raphael Linus Levien and Carlo Adviser-Sequin. *From spiral to spline: Optimal techniques in interactive curve design*. University of California at Berkeley, 2009 (cit. on p. 103).

[56] Yong Liu et al. "Semantic modeling for ancient architecture of digital heritage." In: *Computers & Graphics* 30.5 (2006), pp. 800–814 (cit. on p. 48).

[57] James McCrae and Karan Singh. "Neatening sketched strokes using piecewise French curves." In: *Proceedings of the Eighth Eurographics Symposium on Sketch-Based Interfaces and Modeling*. SBIM '11. Vancouver, British Columbia, Canada: ACM, 2011, pp. 141–148. ISBN: 978-1-4503-0906-6 (cit. on p. 103).

[58] James McCrae and Karan Singh. "Sketch-Based Interfaces and Modeling (SBIM): Sketching piecewise clothoid curves." In: *Comput. Graph.* 33.4 (Aug. 2009), pp. 452–461. ISSN: 0097-8493 (cit. on pp. 103, 121).

[59] James McCrae and Karan Singh. *Sketch-based path design*. Canadian Information Processing Society, 2009 (cit. on p. 103).

[60] Radomir Mech and Przemyslaw Prusinkiewicz. "Visual models of plants interacting with their environment." In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. ACM. 1996, pp. 397–410 (cit. on p. 9).

[61] Erick Mendez et al. "Generating semantic 3D models of underground infrastructure." In: *IEEE Computer Graphics and Applications* 28.3 (2008) (cit. on p. 138).

[62] Paul Merrell, Eric Schkufza, and Vladlen Koltun. "Computer-generated residential building layouts." In: *ACM Transactions on Graphics (TOG)* 29.6 (2010), p. 181 (cit. on p. 48).

[63] Judith Milde and Claus Brenner. "Graph-based modeling of building roofs." In: *AGILE Conference on GIScience*. 2009 (cit. on p. 47).

[64] J Milde et al. "Building reconstruction using a structural description based on a formal grammar." In: *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences* 37 (2008) (cit. on p. 47).

[65] Pascal Müller et al. "Procedural modeling of buildings." In: *ACM Transactions on Graphics* 25.3 (2006), pp. 614–614. ISSN: 07300301. DOI: `10.1145/1141911.1141931`. URL: `http://portal.acm.org/citation.cfm?doid=1141911.1141931` (cit. on pp. 2, 12, 29, 47, 48, 74, 76, 79, 81, 147).

[66] AW Nutbourne, PM McLellan, and RML Kensit. "Curvature profiles for plane curves." In: *Computer-Aided Design* 4.4 (1972), pp. 176–184 (cit. on p. 102).

[67] *Open street map*. URL: `http://www.openstreetmap.org` (cit. on p. 49).

[68] TK Pal and AW Nutbourne. "Two-dimensional curve synthesis using linear curvature elements." In: *Computer-Aided Design* 9.2 (1977), pp. 121–134 (cit. on p. 102).

[69] Yoav IH Parish and Pascal Müller. "Procedural modeling of cities." In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM. 2001, pp. 301–308 (cit. on pp. 8, 9).

[70] Giacomo della Porta. *Wikimedia commons - St. Peter's Basilica*. License: Giacomo della Porta. URL: `https://upload.wikimedia.org/wikipedia/commons/1/15/Petersdom_von_Engelsburg_gesehen.jpg` (cit. on pp. 3, 126).

[71] Adobe Press. *PostScript language reference manual*. Addison-Wesley Longman Publishing Co., Inc., 1985 (cit. on p. 136).

[72] P Prusinkiewicz and A Lindenmayer. *The Algorithmic Beauty of Plants*. 1990 (cit. on p. 9).

[73] Przemyslaw Prusinkiewicz, Mark James, and Radomir Mech. "Synthetic topiary." In: *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. ACM. 1994, pp. 351–358 (cit. on p. 9).

[74] A Schechter. "Synthesis of 2D curves by blending piecewise linear curvature profiles." In: *Computer-aided design* 10.1 (1978), pp. 8–18 (cit. on p. 102).

[75] Robert Schneider and Leif Kobbelt. "Discrete Fairing of Curves and Surfaces based on Linear Curvature Distribution." In: *Curve and Surface Design, Saint-Malo 1999*. Ed. by Pierre-Jean Laurent, Paul Sablonniere, and Larry L. Schumaker. Innovations in Applied Mathematics. Saint-Malo, France: Vanderbilt University Press, 2000, pp. 371–380. ISBN: 0-8265-1356-5 (cit. on pp. 102, 106).

[76] *School of Computer Science and Engineering (SCSE), Nanyang Technological University, Singapore*. URL: http://scse.ntu.edu.sg/Pages/Home.aspx (cit. on pp. 43, 71).

[77] Michael Schwarz and Pascal Müller. "Advanced procedural modeling of architecture." In: *ACM Transactions on Graphics* 34.4 (2015), 107:1–107:12. ISSN: 07300301. DOI: 10.1145/2766956. URL: http://dl.acm.org/citation.cfm?doid=2809654.2766956 (cit. on pp. 2, 14, 79, 81).

[78] George Stiny. *Introduction to shape and shape grammars*. 1980. DOI: 10.1068/b070343. URL: http://goo.gl/9GsWe2 (cit. on p. 74).

[79] George Stiny and James Gips. "Shape Grammars and the Generative Specification of Painting and Sculpture." In: *IFIP Congress (2)*. 1971, pp. 1460–1465 (cit. on pp. 11, 47).

[80] Martin Strobl et al. "Euclides - a JavaScript to PostScript Translator." In: *Proceedings of the International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking (Computation Tools)*. United States: Institute of Electrical and Electronics Engineers, 2010, pp. 14–21 (cit. on p. 137).

[81] Kenichi Sugihara and Yoshitugu Hayashi. "Automatic generation of 3D building models from building polygons on GIS." In: *ICCCBEXI Proceedings of the 11th ICCCBE. Montreal, Canada* (2006), pp. 14–16 (cit. on p. 48).

[82] Kenichi Sugihara and Yoshitugu Hayashi. "Automatic generation of 3D building models with multiple roofs." In: *Tsinghua Science & Technology* 13 (2008), pp. 368–374 (cit. on p. 48).

[83] Franck Taillandier. "Automatic building reconstruction from cadastral maps and aerial images." In: *International Archives of Photogrammetry and Remote Sensing* 36.Part 3 (2005), W24 (cit. on p. 47).

[84] Soon Tee Teoh. "Generalized descriptions for the procedural modeling of ancient east asian buildings." In: (2009) (cit. on p. 48).

[85] Wolfgang Thaller et al. "Creating procedural window building blocks using the generative fact labeling method." In: *ISPRS-International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* 1.1 (2013), pp. 235–42 (cit. on pp. 22–24, 137, 138).

[86] Wolfgang Thaller et al. "Implicit nested repetition in dataflow for procedural modeling." In: *Proceedings of the International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking (Computation Tools), Nice, France*. Citeseer. 2012, pp. 22–27 (cit. on p. 137).

[87] Wolfgang Thaller et al. "Shape grammars on convex polyhedra." In: *Computers and Graphics (Pergamon)* 37.6 (2013), pp. 707–717. ISSN: 00978493. DOI: 10.1016/j.cag.2013.05.012. URL: http://linkinghub.elsevier.com/retrieve/pii/S0097849313000861 (cit. on pp. 16, 18, 47, 73, 75, 76, 81, 136–138).

[88] Yannick Thiel, Karan Singh, and Ravin Balakrishnan. "Elasticurves: exploiting stroke dynamics and inertia for the real-time neatening of sketched 2D curves." In: *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM. 2011, pp. 383–392 (cit. on p. 103).

[89] ToucanWings. *Wikimedia commons - Palace of Versailles*. License: CC BY-SA 3.0/ToucanWings. URL: https://upload.wikimedia.org/wikipedia/commons/9/94/Vue_a%C3%A9rienne_du_domaine_de_Versailles_le_20_ao%C3%BBt_2014_par_ToucanWings_-_Creative_Commons_By_Sa_3.0_-_26.jpg (cit. on pp. 3, 126).

[90] Arie Van Deursen, Paul Klint, and Joost Visser. "Domain-specific languages: An annotated bibliography." In: *ACM Sigplan Notices* 35.6 (2000), pp. 26–36 (cit. on p. 35).

[91] Marco Verch. *Wikimedia commons - Pantheon*. License: CC BY 2.0/Marco Verch. URL: https://upload.wikimedia.org/wikipedia/commons/5/51/Pantheon_in_Rom_%2824200809342%29.jpg (cit. on pp. 3, 126).

[92] Vivek Verma, Rakesh Kumar, and Stephen Hsu. “3d building detection and modeling from aerial lidar data.” In: *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*. Vol. 2. IEEE. 2006, pp. 2213–2220 (cit. on p. 47).

[93] *Virtual Singapore*. URL: http://www.nrf.gov.sg/programmes/virtual-singapore (cit. on p. 141).

[94] DJ Walton and DS Meek. “A controlled clothoid spline.” In: *Computers & Graphics* 29.3 (2005), pp. 353–363 (cit. on p. 103).

[95] Jeff Wilcox. *Wikimedia commons - Neuschwanstein Castle*. License: CC BY 2.0/Jeff Wilcox. URL: https://upload.wikimedia.org/wikipedia/commons/3/37/Neuschwanstein_castle.jpg (cit. on pp. 3, 126).

[96] Peter Wonka et al. “Instant architecture.” In: *ACM Transactions on Graphics* 22.3 (2003), pp. 669–669. ISSN: 07300301. DOI: 10.1145/882262.882324 (cit. on pp. 2, 11, 47, 74, 76).

[97] Rene Zmugg et al. “Deformation-aware split grammars for architectural models.” In: *Proceedings - 2013 International Conference on Cyberworlds, CW 2013* (2013), pp. 4–11. DOI: 10.1109/CW.2013.11. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6680085 (cit. on pp. 18, 20, 75, 77).

[98] Rene Zmugg et al. “Procedural architecture using deformation-aware split grammars.” In: *The Visual Computer* 30.9 (2014), pp. 1009–1019. ISSN: 0178-2789. DOI: 10.1007/s00371-013-0912-3. URL: http://link.springer.com/10.1007/s00371-013-0912-3 (cit. on pp. 18, 20, 73, 75, 77, 136).

List of Figures

List of Tables