Locally context-sensitive Shape Grammars

Markus Eger

Locally context-sensitive Shape Grammars

Master's Thesis

at

Graz University of Technology

submitted by

Markus Eger

Institute for Computer Graphics and Knowledge Visualization (CGV),
Graz University of Technology
A-8010 Graz, Austria

23 May 2013

© Copyright 2013 by Markus Eger

Advisor: Dipl.-Inform. Dr.-Ing. Univ.-Doz. Sven Havemann

Lokal Kontext-sensitive Shapegrammatiken

Diplomarbeit

an der

Technischen Universität Graz

vorgelegt von

Markus Eger

Institut für Computergrafik und Wissensvisualisierung (CGV),
Technische Universität Graz
A-8010 Graz

23. Mai 2013

© Copyright 2013, Markus Eger

Diese Arbeit ist in englischer Sprache verfasst.

Begutachter: Dipl.-Inform. Dr.-Ing. Univ.-Doz. Sven Havemann

Abstract

Shape grammars allow simple modeling of complex structures by providing a small set of rules from which the structures are generated. This allows the fast creation of a variety of similar, yet distinct models. However, modeling interconnections within these structures requires an expressiveness akin to unrestricted formal grammars, which is hard to implement for 3D structures. Furthermore, this approach potentially introduces additional interactions which users might not expect.

In this thesis, a way that mitigates both of these drawbacks is presented by extending the shape grammar formalism with local sub-rules, which can take context into account and therefore model interconnections. By keeping these rules local to parts of the grammar, they are easy to evaluate and provide users with additional expressiveness while preventing unforeseen side-effects.

To demonstrate the feasibility of this approach several examples are presented. These range from simple tile arrangements, which are already hard to describe using only context-free Shape Grammars, to a fully detailed model of the Eiffel Tower, down to the individual rivets.

Kurzfassung

Shapegrammatiken erlauben das einfache Modellieren von komplexen Strukturen mit einer geringen Anzahl von Regeln. Dadurch wird die schnelle Generierung von ähnlichen, jedoch trotzdem unterschiedlichen Modellen ermöglicht. Sobald die Strukturen jedoch in sich verbunden sind, benötigt man eine Ausdrucksstärke die mit der von uneingeschränkten formalen Grammatiken vergleichbar ist, und welche für 3D Modelle nur schwer zu implementieren ist. Außerdem entstehen durch diesen Ansatz potentiell vom Benutzer unerwartete Interaktionen.

In dieser Diplomarbeit wird eine Methode vorgestellt, die diese beiden Nachteile reduziert, indem der Shapegrammatik-Formalismus um lokale Unterregeln erweitert wird, die Kontext in Betracht ziehen können, und daher Verbindungen innerhalb des Modells abbilden können. Da die Anwendung dieser Regeln lokal auf Teile der Grammatik beschränkt wird, wird deren Auswertung erleichtert und dem Benutzer eine zusätzliche Ausdrucksstärke zur Verfügung gestellt, die aber unvorhergesehene Seiteneffekte trotzdem minimiert.

Um die Anwendbarkeit dieses Ansatzes zu demonstrieren, werden einige Beispiele vorgestellt. Diese reichen von einfachen Fliesenanordnungen, die aber mit kontext-freien Shapegrammatiken bereits schwer zu modellieren sind, bis hin zu einem detaillierten Modells des Eiffelturms, der bis zu den einzelnen Nieten modelliert wurde.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Place	Date	Signature

Eidesstattliche Erklrung

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

Ort	Datum	Unterschrift

Preface

Writing a thesis is a major undertaking, and this one was no exception. The last one and a half years of my life were strongly influenced by bringing the Eiffel Tower to ”life”, but I would not have been able to achieve this without the supervision of Dr. Havemann, and I’m grateful for his patience and guidance. The other people at CGV, especially Ulrich Krispel, Wolfgang Thaller and Rene Zmugg, also have my gratitude for discussing ideas with me and showing me how to work with the existing implementation.

Of course, the help from professionals is only half the story. The other half is the ongoing support from my family, who supported me through all my studies. I also have to thank my friends, whom I had to neglect more often than I would have liked to get more work done, but they were always understanding and supportive. This thesis wouldn’t have been possible without all of them.

Contents

Preface xiii

Contents iii

List of Figures vi

List of Tables vii

1 Introduction 1
- 1.1 Structure of this Document . . . 2

2 Related Work 3
- 2.1 Shape Grammars . . . 3
  - 2.1.1 Split Grammars . . . 6
  - 2.1.2 Modeling Interconnected Structures . . . 6
  - 2.1.3 Applications of Shape Grammars . . . 7
  - 2.1.4 Procedural Modeling without Shape Grammars . . . 8
- 2.2 Generative Modeling Language . . . 9
  - 2.2.1 GML Execution . . . 9
  - 2.2.2 GML Example . . . 9
  - 2.2.3 Dictionary Stacks . . . 11
  - 2.2.4 Convex Polyhedra . . . 11
  - 2.2.5 Modeling Environment . . . 12
- 2.3 Split Grammars in GML . . . 12
  - 2.3.1 Representing Grammar Rules in GML Code . . . 13
  - 2.3.2 Simple Example: Cube Skeleton Grammar in GML . . . 14
  - 2.3.3 Limitations of the GML Split Grammar Implementation . . . 15

3 Extending Shape Grammars with Context-Sensitive Sub-Rules 17
- 3.1 Approaching the solution . . . 17
  - 3.1.1 Approach 1: Using 3D Space as ”Storage Container” . . . 17
  - 3.1.2 Approach 2: Defining and Using Attaching Points . . . 18
  - 3.1.3 Approach 3: Exchanging Information between Non-Terminal Symbols . . . 18
  - 3.1.4 Approach 4: Using Optimization Algorithms on Split Parameters . . . 19
  - 3.1.5 Towards Context-Sensitive Sub-Rules . . . 20

3.2 Context-Sensitive Sub-Rules . . . 21
3.2.1 Definition . . . 21
3.2.2 Evaluation of the Sub-Rules . . . 22
3.3 Extended Grammar Operations . . . 22
3.3.1 List of Grammar Operations . . . 23
3.3.2 Variable Number of Result Scopes . . . 24
3.3.3 Combining Grammar Operations . . . 25
3.3.4 Extending the Set of Grammar Operations . . . 25
3.4 Scope Parameters . . . 25
3.4.1 Defining Scope Parameters . . . 25
3.4.2 Scope Parameter Inheritance . . . 26
3.4.3 Scope-Local Coordinate Systems . . . 26
3.4.4 Inheriting from Multiple Scopes . . . 26
3.4.5 Semantic Level-of-Detail . . . 27
3.5 Nondeterminism in the Evaluation of the Grammar . . . 27
3.6 Substructure Interfaces using Labeled Non-Terminal Symbols . . . 27

4 Implementation **29**
4.1 Extensions to the Split Grammar Implementation . . . 29
4.1.1 Radial Split . . . 29
4.1.2 Scope-Local Coordinate Systems . . . 30
4.2 Context-Sensitive Sub-Rules . . . 31
4.2.1 Defining Rule Context . . . 31
4.2.2 Setting Scope Labels . . . 32
4.2.3 Implementing Lazy Evaluation of the Grammar . . . 32
4.2.4 Scope Selection and Sub-Rules . . . 32
4.3 Grammar Operations . . . 33
4.3.1 List of Added Grammar Operations . . . 33
4.4 Scope Parameters . . . 45
4.4.1 Setting and Getting Scope Parameter Values . . . 45
4.4.2 Scope Parameter Inheritance . . . 45

5 Example Models **47**
5.1 Simple Examples . . . 47
5.1.1 Grid Tiles with Connections . . . 47
5.1.2 Wooden Planks with Nails . . . 48
5.1.3 Grid Spanning Multiple Separate Areas . . . 49
5.1.4 Contour Example - Convexify . . . 52
5.1.5 Contour Example - Decorate . . . 52
5.2 Eiffel Tower . . . 54
5.2.1 General Structure . . . 54
5.2.2 Single Leg Interface . . . 54
5.2.3 Support Beams with Lattice Structure . . . 55

5.2.4 Connecting Bars with Rivets 56
5.2.5 Placing Arches 56
5.2.6 Lower Platform 57
5.2.7 Higher Platform Support Structure 58
5.2.8 Common Grid in Top Part 59
5.2.9 Campanile 59
5.2.10 Evaluation of the Model 60
5.2.11 Eiffel Tower Variations 62
5.2.12 Level-of-Detail 62

6 Conclusion **65**
6.1 Summary 65
6.2 Limitations 65
6.3 Future work 66

List of Figures

2.1 A simple shape grammar definition . . . 4
2.2 Shape grammar derivation . . . 4
2.3 A more complex shape . . . 5
2.4 The subshape problem . . . 5
2.5 A support grid of the Eiffel Tower . . . 7
2.6 The GML Studio IDE . . . 12
2.7 Evaluation of the cube skeleton grammar . . . 15
2.8 Tree vs. semi-lattice . . . 16

3.1 Connecting columns on a landscape . . . 18
3.2 Message passing example . . . 19
3.3 Optimizing split parameters . . . 20
3.4 A support grid of the Eiffel Tower . . . 21

4.1 Result of a radial split . . . 31
4.2 Result of the `connect-convex` grammar operation . . . 34
4.3 Result of the `connect-union` grammar operation . . . 36
4.4 Result of the `connect-intersection` grammar operation . . . 37
4.5 Result of the `connect-intersection` grammar operation . . . 38
4.6 Result of the `extend-infinite` grammar operation . . . 39
4.7 Result of the `extend-scope` grammar operation . . . 40
4.8 Result of the `connect-rod` grammar operation . . . 41
4.9 The problem of coincident planes . . . 42
4.10 ”Find contour” example . . . 43
4.11 Finding contours in vertices with valence greater than 2 . . . 44
4.12 ”Convexify” example . . . 46

5.1 Cell grid example . . . 48
5.2 Planks connected with nails . . . 50
5.3 Multiple aligned grids . . . 51
5.4 Fitting an aligned grid into holes between turnstile grid cells . . . 53
5.5 Decorating a contour outline . . . 53
5.6 The Eiffel Tower model . . . 54
5.7 Interface of a single leg of the Eiffel Tower . . . 55

5.8 Main support beams with lattice structure 56
5.9 Rivets for Eiffel Tower connections 57
5.10 One of the decorative arches of the Eiffel Tower, also note the lower platform, and the grid that connects the arches with the lower platform 57
5.11 The lower platform of the Eiffel Tower 58
5.12 The supporting structure for the higher platform 58
5.13 Shared grid between the top parts of the Eiffel Tower 59
5.14 The campanile on top of the Eiffel Tower 59
5.15 Comparison of the Eiffel Tower model with a blue print and a photograph 60
5.16 Comparison of the Eiffel Tower model with a blue prints of a pillar part 60
5.17 Comparison of the Eiffel Tower model with a photograph from below 61
5.18 Variants of the Eiffel Tower 63
5.19 The different levels of detail of the Eiffel Tower 64

List of Tables

5.1 Summary of required parameters for the leg 56

Chapter 1

Introduction

The creation of 3D models for the use in virtual environments has become a more and more important task in recent years due to both the increasing size of these environments and the required level of detail. On the other hand, the manual creation of such models by the use of 3D modeling tools like Maya [4] or 3DS Max [3] has not decreased as much in complexity and effort. This means, the traditional approach of manual modeling does not scale well to the current and especially future requirements. As an alternative, procedural modeling allows the creation of detailed models, which can then be used in more ways than models created with traditional tools, for example when different levels of detail are needed, or only parts of the model are required. It is also possible to create variations of a model easily, for example to place multiple similiar houses in a city, without requiring to have each one modeled individually.

Several different systems for procedural modeling have emerged over the years, for example *L-Systems*, which model the organic growth of plants, or *shape grammars*, which are able to produce complex shapes by simple replacement rules. A specialization of shape grammars, named split grammars, operate on named *scopes*, splitting them into sub-scopes and continuing on these sub-scopes. This recursive "split-and-process" approach, which in essence makes the grammar *context-free*, solves the complexities in the evaluation of general shape grammars, but also puts restrictions on the expressibility on them. Since all sub-scopes are processed independently from each other, different parts of the model, once split, have no way of communicating with each other any more. However, real, complex structures require interconnections between model parts, starting from rivets to hold parts together to wider spanning connections like bridges. There are a few approches to add restricted forms of context-sensitivity to split grammars, but since they are either mostly concerned with the prevention of occlusion, like CGAShape [33], or only provide the ability to add connections while ignoring other possibilities context-sensitivity could provide, like described in [26], there is still room for improvement.

This thesis describes a new approach to the problem, by adding true context-sensitivity to split grammars, but restricting it to subtrees and logical rather than geometric connections. This solves issues with the complexity of the evaluation and use, while still providing much greater expressiveness.

1.1 Structure of this Document

Chapter 2 gives an overview over the rich history of shape grammars, describes the recently introduced split grammars and discusses various implementations. The implementation basis of this thesis, a split grammar implementation written in the Generative Modeling Language (GML) is also described.

In chapter 3 the extension to split grammars is presented, which introduces context-sensitive sub-rules, and allows the modeling of interconnections between sub-structures. The extended semantics are described and further implications of this extension are also discussed.

Chapter 4 presents the actual implementation of this extension.

To demonstrate that the presented approach is actually capable of modeling complex structures, chapter 5, shows several example structures modeled with the new approach. The highlight of this demonstration is a detailed model of the Eiffel Tower, down to the individual rivets. Additionally, a few variations of the Eiffel Tower are shown.

Finally, chapter 6 discusses the limitations this new approach still has and what further steps could be taken to overcome them.

Chapter 2

Related Work

2.1 Shape Grammars

Shape grammars were introduced by George Stiny and James Gips in 1971 [43], and are related to formal grammars for languages as described by Noam Chomsky [9]. They consist of a set of terminal symbols, a set of non-terminal symbols, a set of replacement rules, and a start shape, also called ”seed shape” in the literature [41]. Non-terminal and terminal symbols all consist of shapes, which were defined as a finite set of straight lines in the 2D plane in the original formulation, but the mechanism works equally well for higher dimensions and non-straight lines.. Each rule consists of a left-hand side and a right-hand side, where the left-hand side is any combination of elements of the set of terminal-symbols and the set of non-terminal symbols. Figures 2.1 to 2.3 demonstrate a simple shape grammar. The application of the grammar then matches all rules' left-hand sides against all sub-shapes, taking any necessary geometric transformations into account and if a match is found, it is replaced by the corresponding right-hand side, which is also transformed in the same way. This application is then repeated on the new shape until no more non-terminal symbols are present. In the seminal paper, the evaluation was actually performed by drawing the shapes and performing replacements for the next drawing by hand [15]. The first computer program to evaluate shape grammars, written by James Gips in 1975, ignored the problem of finding matching emerging sub-shapes. Using the non-terminal symbols as markers, the derivation process can be influenced to a great extent. Rule-sets which differ only in these markers can result in quite different shape languages [40].

After that, shape grammars have been extended in several ways. One such extension has been to allow rules to be parameterized, resulting in rules schemas, from which concrete rules can be obtained by assigning values to the free variables [41]. The general problem of matching emerging sub-shapes is computationally hard. This problem is show in figure 2.4. Therefore, many shape grammar implementations opt to implement it only partially or not at all, leading to alternative formulations with restricted functionality, like set-based grammars [30]. To mitigate this problem, research into the necessary data structures and algorithms to solve the sub-shape problem was performed by Ramesh Krishnamurti [27], [28] resulting in a shape grammar interpreter with subshape detection for 2D shapes [29]. As the

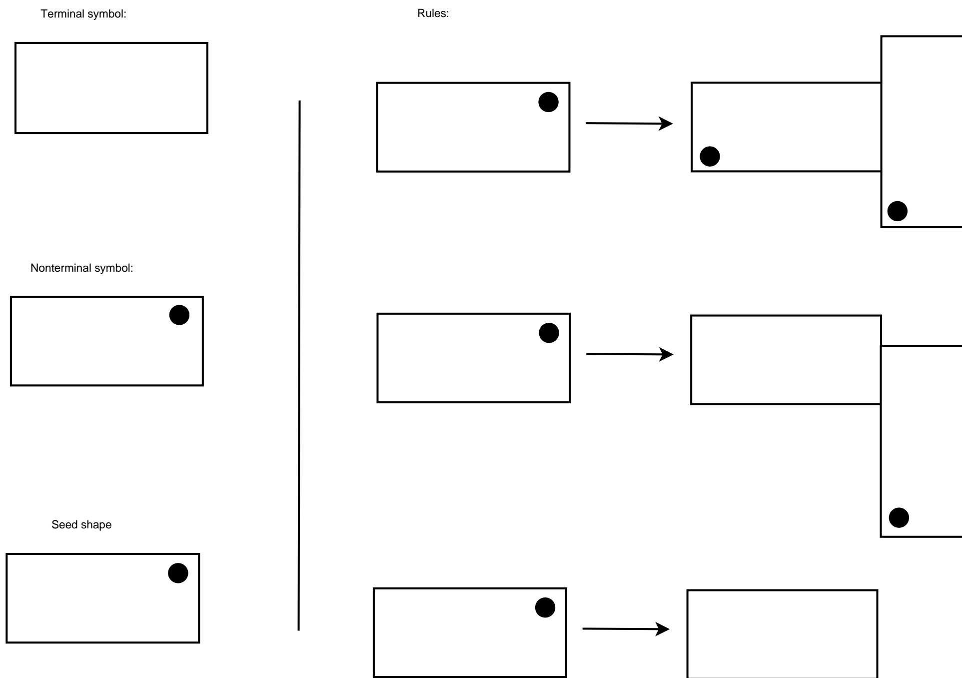


Figure 2.1: The definition of a simple shape grammar, consisting of only one terminal and one non-terminal symbol. The three rules can only apply to non-terminal symbols. The ”dot” is used to provide a ”marker” for non-terminal symbols.

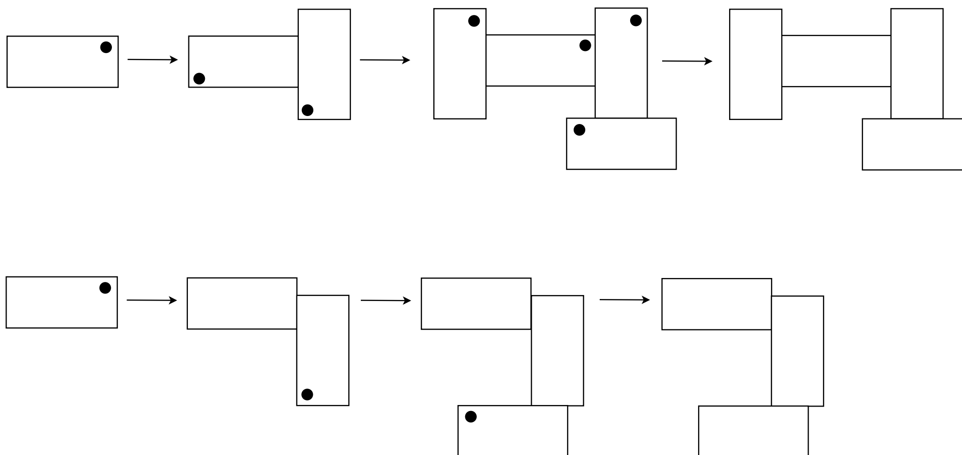


Figure 2.2: Two different derivations using the same shape grammar. In the first, the first rule is applied, resulting in two new non-terminal symbols, to which the first rule is then applied again. By applying the third rule to all non-terminals then terminates derivation. The second example uses the second rule twice instead, resulting in a noticeably different shape.

sub-shape problem is even harder for three-dimensional shapes, none of the 3D shape grammar implementation surveyed in [15] fully supports arbitrary emerging sub-shapes. It took until 2004 until such an interpreter was finally implemented [7], which also supports curved line shapes, but it relies on user input for parameterized rules making its use for complex structures difficult. A step in a different direction was taken with the introduction of plit grammars, described in more detail below. Numerous other extensions have been proposed over the years, like colors [25] or weights [42], which amount to assigning different

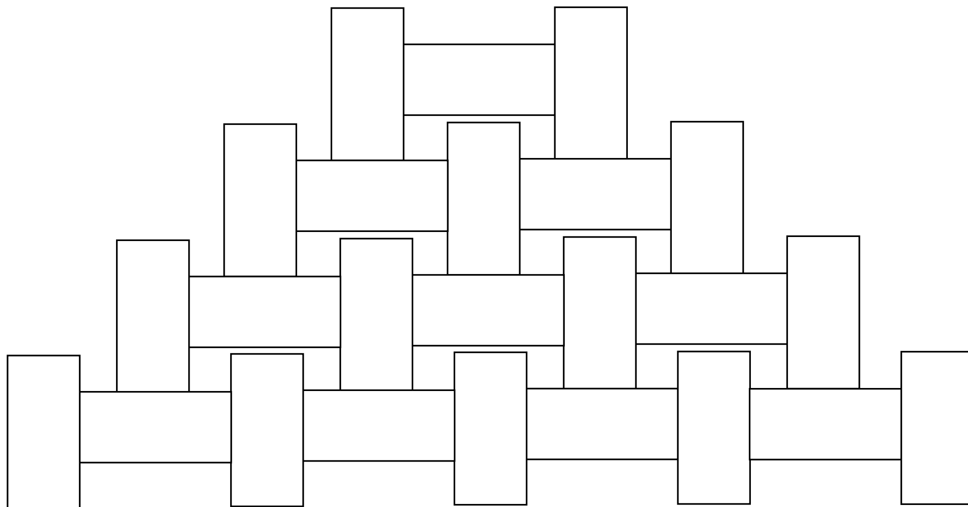
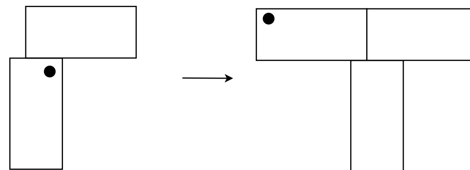


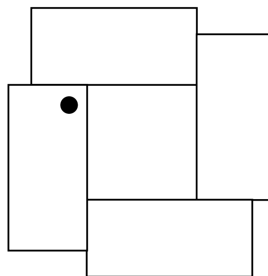
Figure 2.3: Another sample element of the shapes generated by the same shape grammar.

attributes to shapes.

In [15] Gips raises the questions which of these extensions are actually useful, and which ones are



(a) An additional rule is added to the shape grammar example from above



(b) After three applications of the third rule, a shape matching the left hand-side of the new rule appears. However, it consists not only of the newly inserted part, but also of one that was already there before. In fact, that other part is a terminal symbol and could so far be "ignored" for rule evaluation. The "subshape problem" is how to keep track of all these inserted shapes in combination with the newly inserted ones.

Figure 2.4: The subshape problem

even vital to make shape grammars as a whole useful for the industry. He also mentions several other issues, including the already discussed difficulty of dealing with emerging sub-shapes. Another point he raises is the question of usability, since most shape grammar systems require very detailed knowledge of their innards to be used effectively, an opinion that is also shared by Terry Knight in [24]. In this paper,

Knight also discusses the problem of decidability for shape grammars with different restrictions on their rules structure.

2.1.1 Split Grammars

Split Grammars were introduced in 2003 by Wonka et al. [48] as a specialization of shape grammars. They operate on ”scopes”, which are labeled entities with an associated box as its geometry. A ”split” operation is then defined as one that decomposes that box into several smaller boxes, thereby creating new scopes. Each grammar rule can perform such a split into any number of pieces along one of the boxes principal directions, resulting in a derivation tree of scopes and sub-scopes. Here, *non-terminal symbols* are the named scopes, while *terminal symbols* replace a scope with actual geometry that is placed in 3D space.

This approach has the advantage that emerging sub-shapes are not an issue, since the rules operate on labels rather than on the shapes themselves. However, since each split is performed only on a single scope, they are inherently *context-free*. For many tasks in architecture, this is not inherently a problem, since the split-process often models how architects work quite naturally [32]. Müller et al. show how they use CGA Shape grammar system, which is based on Split Grammars, to model a wide variety of buildings for a cityscape [33]. Of particular note is their ability to take context into account, to prevent windows from being placed where adjacent building parts are located, but this is limited to the special cases of occlusion checks, and the ability to define snap lines, as to align features, like windows and along global axes.

Another implementation, that also serves as the basis for this thesis, was presented by Havemann et al, and integrates shape grammars with imperative modeling [18]. Instead of writing grammar rules in a declarative fashion, they are directly encoded in an imperative fashion in a programming language. This allows the use of the full power of the underlying programming language where necessary, while still preserving the basic properties of shape grammars otherwise. In [19], this approach is also elaborated upon by showing how the use of convex polyhedra makes modeling complex architecture more natural. Section 2.3 describes this approach in more detail.

2.1.2 Modeling Interconnected Structures

As described above, split grammars are inherently context-free, which poses certain limitations on what can be modeled. In man-made structures, interconnections happen naturally because the structural integrity of the structure requires nails, rivets or other kinds of connectors to be placed. Also, the construction process itself lends to the separate construction of distinct parts that are connected at some later point, like building a bridge between different buildings, or placing trolley wires over a street, which is attached to the buildings. Another subtle type of connection exists when there is an alignment between parts of the model, for example grid bars in distinct grids that are aligned. An example of this, as it occurs in the Eiffel Tower, is shown in figure 2.5. In CGAShape, as described by Müller et al [33], there is some support for context-awareness. Specifically, certain planes can be used as so-called ”snap lines”. This

Figure 2.5: A photograph of part of the Eiffel Tower (Released under CC-BY-SA 3.0 by Benh Lieu Song) in which the support grid of one of the platforms of the Eiffel Tower is shown. Note how the grid bars align over the distinct grids.

means, they can be stored at some point in the grammar, and re-used at another. For a number of use-cases this already provides sufficient control over placement in otherwise unrelated parts of the model, and does solve the problem of grid alignment. Since the application is automatic, it does not have to be taken into account when modeling both sub-structures. However, at least one the parts has to be aware of this connection, and has to provide the snap lines. It is therefore impossible to take two completed models, put them together in the same scene and simply add snap-lines. Other types of interconnections, like bridges are also not supported at all, unless they are modeled explicitely from the start.

Another approach to this problem is presented by Krecklau et al in their paper ”Procedural Modeling of Interconnected Structures” [26]. It allows each sub-structure to define possible points of attachment and then take these ”attaching points” of different structures and connect them using direct matching or even geometric queries. The implementation works by putting the attachment points into arrays, that are simply passed as additional parameters to the rules by reference. So, while the basic concept is sound, and allows for a great degree of flexibility, it is not truely integrated with the theory of shape grammars. It also does not solve the problem of aligning grids. The approach described in this thesis is strongly influenced by the idea of attaching points, but integrates them with the general idea of shape grammars rather than processing them separately, which results in additional possibilities like solving the grid alignment problem.

2.1.3 Applications of Shape Grammars

There are several applications of shape grammars [6]. The first is called *generation*: By performing the derivation a shape can be generated, and, if the derivation is non-deterministic, this may be used to generate completely new shapes that just share their basic structure with others generated by the same structure. One practical application of this is in the creation of computer games or other virtual environments [47], where creating diverse, yet similiar 3D models often consumes a large part of the budget. Most research into shape grammars has been performed for this application, especially for the generation of various building models, for example for Queen Anne houses [11], Palladian architecture [44] or mass customized housing [10]. However, shape grammars have also been used to describe the structure of

paintings [23], coffee machines [1] or soft drink and shampoo bottles [7].

It is also possible to use shape grammars to *parse* a given shape and determine if it falls into the class of shapes described by the given grammar, similiar to how a formal grammar can be used to parse a sentence. This can, be used to classify shapes, or simply determine whether a given shape fulfills the required properties. For example, in [45], the authors describe how they use shape grammar parsing via reinforcement learning to segment a facade given a picture.

Another application of shape grammars involves their *synthesis*. Given a number of shapes, an algorithm can determine a shape grammar that describes all these shapes. The resulting grammar can then be used for generation and parsing as described above. Mathias et al describe how they use shape grammars, complemented by Structure-from-Motion to reconstruct 3D buildings [31].

2.1.4 Procedural Modeling without Shape Grammars

Shape grammars are not the only generative system capable of generating complex structures. Several other systems exist, but they either specialize in the creation of natural shapes, and are therefore usually unsuited for the creation of man-made structures, or they are too general and complex to be used for inverse modeling. This section shall give a short overview of the alternatives.

For landscape generation, the most commonly used approach are fractal noise generators, as described in [14]. This base approach is then often refined to make the result look more realistic by eroding parts to account for physical phenomena like rainfall erosion or material properties [34]. There has also been research into how to incorporate rivers in the creation of heightmaps, both by starting with a river network and creating mountain ridges based on them [21], and by placing rivers during the creation of the heightmap based on where mountains are placed [5]. These approaches are clearly not applicable to rigid building structures, since they focus on the creation of landscapes, which are fundamentally different from man-made structures.

Another generative model are L-Systems. They are a rewriting system similiar to shape grammars but instead of using shapes, they originally worked with strings of symbols instead, ignoring geometry. The resulting string is then used for drawing, for example with turtle graphics [36]. Later extensions have added geometry-awareness, which blurs the distinction between L-Systems and shape grammars. L-Systems are particularily well suited for modeling the natural growth of plants, which is also the field where they originated from [37]. There are, however, also applications of L-Systems for man-made structures like cities [35]. While it is also possible to formulate these structures using shape grammars, the simplicity of L-Systems gives them the advantage in this field, also allowing them to be used for inverse procedural modeling more easily [39]. However, once building models grow more complex, ordinary L-Systems are no able suited to model them, and the more complex shape grammars are better suited.

On the other end of the complexity scale reside shape description languages like the Generative Modeling Language (GML), which is described in detail below, or simply using any programming language to generate shapes. For example, the game ".kkrieger" procedurally generates a full 3D level, several models and complex textures with a 96 KiB executable written in C++ and x86 assembly [46]. However, this code is highly specialized and can not easily be adapted, nor generated by inverse modeling tasks.

2.2 Generative Modeling Language

The Generative Modeling Language (GML) is a concatenative, stack-based language, which shares its syntax almost entirely with Postscript, but contains operators for 3D shapes rather than for typesetting [17]. It is therefore a language for the description of three-dimensional shapes, with a runtime environment consisting of several libraries for e.g. splines, convex polyhedra or integration with OpenSG. As a programming language with very simple rules and almost no syntax, but including all these libraries for shape representation, it is very well suited for encoding shape grammars. In fact, a split grammar implementation in GML was already available and served as the implementation basis for this thesis. This implementation is described in detail in section 2.3, but first a short overview of GML is given, which is essential for understanding the implementation.

2.2.1 GML Execution

A GML program consists of a stream of tokens, which are either data or operator names. Execution consumes tokens from this stream and performs an action that depends on their type. Data is pushed on an operand stack, while names are looked up in a dictionary mapping from names to arbitrary objects and the found value is then executed. These values may be functions which can then consume values from the stack and/or push new ones. The token "`[`" is pushed onto the stack like any literal, while the token "`]`" pops values from the stack until a "`[`" is found, and pushes an array containing all popped values. The tokens "{" and "}" proceed in the same way, except that they result in a function (an "executable array"). Tokens starting with a slash are interpreted as literal names and simply pushed onto the stack. The function "`def`" expects a literal name and an arbitrary object on the stack and then adds that object under the given name to the dictionary. The "dictionary" itself is actually a stack of dictionaries, onto which a new one can be pushed with "`begin`" and the top one can be popped with "`end`". A name lookup then starts with the top-most dictionary on this dictionary stack, and if the name being looked for is not found, the one below is searched. Search proceeds downwards the dictionary stack until the name is found, or the bottom is reached, in which case a `NameError` is emitted. By pushing new dictionaries onto the stack, it is possible to provide scopes for names. This name lookup in dictionary stacks is a central point in the evaluation of the grammar, where non-terminal symbols are associated with names in the GML implementation. A more detailed description of GML can be found on the GML homepage [12], which also contains a wiki with a reference for all built-in operators [13]. Only the features relevant to this thesis will be elaborated upon here.

2.2.2 GML Example

```
/f 1 def
dict begin
/f
{ 2 add }
def
3 f
end
3 f
```

Execution proceeds as follows:

- Line 1 first pushes the name `f` on the operand stack, followed by the value 1. Then, the operator `def` pops both and adds an entry for the name `f` with the value 1 to the default dictionary

- Line 2 then creates a new, empty dictionary on the operand stack, which is consumed by the `begin` operator, which pushes it on the dictionary stack

- Line 3 simply pushes the name f on the operand stack

- Line 4 creates an executable array consisting of the tokens `2` and `add` on the operand stack. When a { is encountered, the interpreter is put into a so-called ”deferred” mode, where it does not interpret subsequent tokens until the executable array is completed. Otherwise, the `add` operator would be looked up and executed immediately, which is undesired. Only when the executable array is finally executed, interpretation proceeds normally.

- Line 5 then takes the name `f` pushed in line 3 and the executable array pushed in line 4 from the operand stack, and puts them in the top-most dictionary of the dictionary stack. Since line 2 pushed a new dictionary, the entry defined in line 1 is not ”lost”.

- Line 6 first pushes the value 3 on the stack, then the name `f` is looked up in the topmost dictionary on the dictionary stack. Since it does contain an entry for that name, the value associated with it is executed. This value is the executable array from line 4. To execute the array, its tokens are executed in order, which means that first 2 is pushed on the operand stack, and then the operator `add` is executed. This built-in operator takes the top two values from the operand stack, in this case 2 and 3 and pushes their sum, in this case 5.

- Line 7 simply discards the top dictionary from the dictionary stack.

- Line 8, finally, pushes 3 on the operand stack again, and then performs a lookup for the name `f`. Since the dictionary that contained the function was just discarded, the value that is found is actually the one defined in line 1, which is 1 and therefore simply pushed on the operand stack.

- In the end, the operand stack contains, from bottom to top, the values 5, 3 and 1.

2.2.3 Dictionary Stacks

A dictionary stack, as used by GML for name lookups, can also be used in a more general setting, since every inheritance relation for named values follows this pattern. For example, the derived classes in object-oriented programming languages can override names defined by their parent class, but if they do not, the one defined in the parent class is used. A similiar situation occurs in the derivation of shape grammars, where each sub-scope will inherit properties of its parent scope.

GML provides dictionary stacks also as actual objects that can be created and used. A new dictionary stack can be created with `dictstack`, and dictionaries pushed to and popped from it with `dictstack-begin` and `dictstack-end`. When a name is looked up in a dictionary stack object, it is first looked up in the topmost dictionary, then, if it is not found there, in the one right below and so on, analogous to the interpreter's dictionary stack. A dictionary stack object can also be pushed onto the interpreter's dictionary stack, and then behaves like all its contained dictionaries had been pushed in order, except it can also be popped again as a single entity. One useful property of dictionary stacks is that they refer to the contained dictionaries by reference, therefore it is possible to have the same dictionary contained in multiple dictionary stacks. This leads to a useful pattern to implement ”inheritance”: The base object is a dictionary stack with a single dictionary. The derived object is a dictionary stack, too, that contains a reference to the dictionary in the parent's dictionary stack, and a new dictionary on top of it. This allows the derived object to define its own values for any attribute, or simply use the one of the parent.

2.2.4 Convex Polyhedra

As noted above, split grammars originally operated on box-shaped scopes, which limited their expressibility. It is, however, straightforward to extend the principle to more general shapes, like convex polyhedra. Every split operation on a convex polyhedron that uses planes as splitters results in more convex polyhedra. Since convex polyhedra can be represented compactly they are an ideal choice for the use in shape grammars.

The GML runtime environment contains a library of functions for the manipulation of convex polyhedra. It represents a convex polyhedron as the intersection of any number of halfspaces, each of which is represented by a directed plane. To avoid numerical problems with floating point numbers, planes are actually represented in integer coordinates. For this, the three-dimensional space is covered with a three-dimensional grid, where each grid node can be represented in integer coordinates. A plane is then described as a node plus a normal vector in that node. The actual size and the center of the grid can be set up with the function `cp-setupgrid`.

One subtely to note with this representation is that a point in three-dimensional space can be represented as the intersection of three planes. However, since planes can also intersect in locations that are not aligned with grid points. This means that not every point that can be obtained as the intersection of three planes can also be represented exactly in integer coordinates. To preserve accuracy, points are therefor represented using three planes where necessary. However, this means, that given two points rep-

resented as the intersection of three planes each, it is not always possible to represent a plane on which both of these points lie. Care must therefore be taken to only use the planes that already exist if accuracy is desired.

2.2.5 Modeling Environment

Of course it is possible to write GML programs in a simple text editor and then execute them a standalone interpreter, but a text editor is not the ideal development environment for 3D modeling. In fact, the IDE that was used, GML Studio, consists of two windows: One for editing text and a 3D viewer that immediately shows the results of any change that is made. This approach allows a fast development cycle, and immediate correction of potential problems.

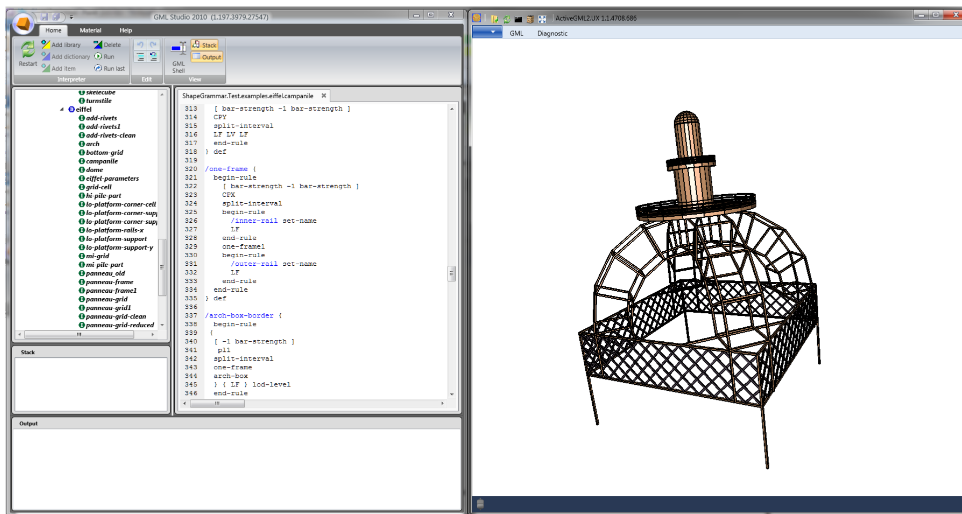


Figure 2.6: The GML Studio IDE

2.3 Split Grammars in GML

As described above, an implementation of Split Grammars was the basis for this thesis. It is implemented in GML, and each grammar rule is also represented by a GML function. To apply a rule therefore means to call the corresponding function, which means the "grammar tree" will be represented by the call tree. A rule will state how to split the current scope and then call other rules, or terminal symbols. The two main terminal symbols, used as leaves of the grammar tree, are `terminal-fill`, to place a convex polyhedra at the scopes location, and `terminal-void` to avoid that. The short-hand names `F` and `V` are also defined for `terminal-fill`, or `terminal-void`, respectively.

Evaluation works as follows: The system maintains a stack of scopes, where the top of the stack is the current scope that all rules operate on. At the beginning a single scope, representing the seed scope is pushed onto that stack. Each split-operation contained in the grammar rules then simply takes the top of the stack, splits it into several new scopes and pushes those onto the scope stack again. These scopes are then operated on by the rules and terminal symbols called by the currently executed rule. Rules pro-

ceed in the same manner recursively, but when a terminal symbol is encountered, the top of the stack is consumed by it.

Scopes are represented as simple dictionaries, and each sub-scope inherits all entries from its parent, but may modify them subsequently. One important set of parameters of each scope is its coordinate system. By having it local to the scope, the same sub-structure may be built with the same rules, even if the scopes to which they are to be applied have different orientations.

The basic grammar operation is called `split-interval`. It takes an array of real numbers and an arbitrary direction as parameters and splits the current scope along that direction into as many sub-scopes as the array has elements. For each array entry that is positive, the size of the created sub-scope in the given direction is chosen to be that value, while negative values are used for relative splits. This works as follows: First, the width of the parent scope in the given direction is calculated, and all absolute sizes are subtracted from that. Then the remainder is divided by the sum of the relative sizes, which gives the size for one "unit" of relative size. Then, for each sub-scope that shall have relative size, that unit size is multiplied with the the given relative size to get the absolute size. This means, for example, that the interval `[ -1 -1 -1 ]` will split a scope into three equally sized pieces, while `[ -1 1 -1 ]` will split it into one part of absolute width 1 in the middle, and two equally sized parts on either side.

2.3.1 Representing Grammar Rules in GML Code

Each grammar rule is encoded as a GML function. This function is named like the non-terminal symbol on the left hand-side of the rule, performs the necessary split operation and then calls the necessary other non-terminal and terminal symbols. This means that evaluation is always deterministic, since only the function which is called for a particular non-terminal symbol is uniquely identified.

A grammar rule can be written as:

$$L \rightarrow R\ [fn]$$

Where L is a single non-terminal symbol, fn describes the grammar operation to be performed, and R is a number of terminal and non-terminal symbols corresponding to the number of result scopes of the grammar function. For example, a rule to perform a split in X-direction on a non-terminal symbol named M, resulting in three pieces, one empty space with width 1.0 in the middle and the others filled and equally distributed over the remaining width in X-direction would be written as:

$$M \rightarrow F\ V\ F\ [\text{split-interval}([-1, 1, -1], X)]$$

In GML, the same would be achieved by creating a function named M, that performs the split and calls the appropiate terminal symbols:

```
1  /M {
2     [ -1 1 -1 ]
3     CPX
```

```
4     split-interval
5     F V F
6 } def
```

Note that CPX refers to the X-direction of the current scope. To use the global X-direction, WPX would have to be used instead.

Of course, to use the grammar, a seed scope has to be provided and the correct start rule be called. This works by calling the scope-box function, that expects two opposing corners of a box and a material number on the stack and pushes a new scope onto the stack, that uses exactly this box as its geometry. Then, the function representing the start rule can simply be called.

2.3.2 Simple Example: Cube Skeleton Grammar in GML

To demonstrate how a grammar rules correspond to their representation in GML, a simple grammar, that generates a cube skeleton is shown. This grammar is given as:

$$
\begin{aligned}
A \rightarrow & B\ C\ B\ [\text{split-interval}([0.05, -1, 0.05], X)] \\
B \rightarrow & F\ D\ F\ [\text{split-interval}([0.05, -1, 0.05], Y)] \\
C \rightarrow & D\ V\ D\ [\text{split-interval}([0.05, -1, 0.05], Y)] \\
D \rightarrow & F\ V\ F\ [\text{split-interval}([0.05, -1, 0.05], Z)]
\end{aligned}
$$

As described above, a GML function is created for each rule. This function performs the necessary split operation and then calls the rules representing the right hand-side of the grammar rule. Additionally, the seed shape is created as a unit cube, and the start rule, in this case A applied to it. In GML this would be written as:

```
1  /A {
2    [ 0.05 -1 0.05 ] CPX split-interval
3    B C B
4  } def
5
6  /B {
7    [ 0.05 -1 0.05 ] CPY split-interval
8    F D F
9  } def
10
11 /C {
12   [ 0.05 -1 0.05 ] CPY split-interval
13   D V D
14 } def
15
16 /D {
17   [ 0.05 -1 0.05 ] CPZ split-interval
18   F V F
```

```
19 } def
20
21 (0,0,0) (1,1,1) 5 scope—box
22 A
```

Figure 2.7 shows how this code is evaluated step by step.

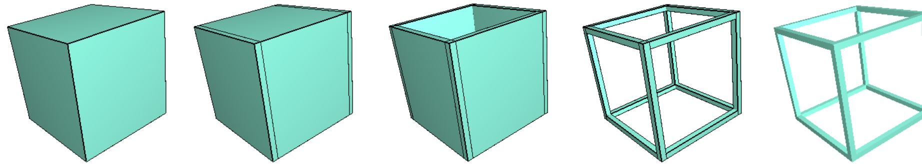


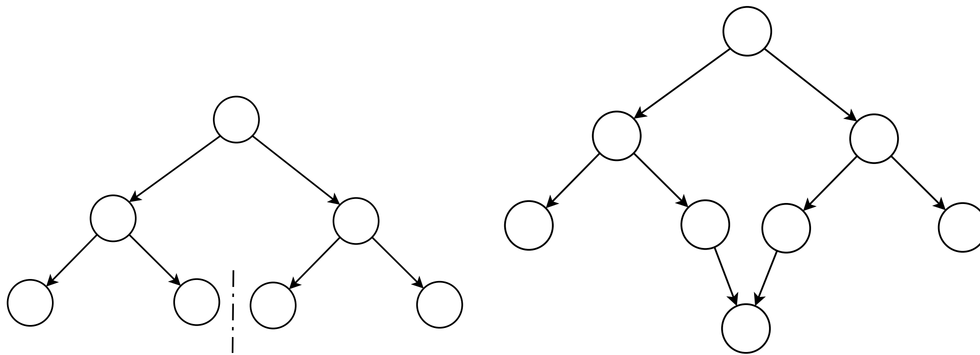
Figure 2.7: Evaluation of the cube skeleton grammar

2.3.3 Limitations of the GML Split Grammar Implementation

The existing system, while powerful, poses certain restrictions on what can be modeled. Chief among them is the difficulty when modeling connections between different parts of the created structure. Rule calls proceed in a tree-like fashion, and siblings in this tree do not know anything about one another. The problem with this is that many man-made structures would require this knowledge, as described in Christopher Alexander's famous piece "A city is not a tree" [2]. In it, the author argues, from the point of view of a designer, that a living city has interactions between its underlying hierarchical structure. The "tree" referred to in the title is not a plant, but rather the data structure, and the author argues that a city is better modeled as a semi-lattice to account for these interactions. As an example, the paper lists a drug store next to a traffic crossing, which has a newspaper stand outside. Clearly, the newspaper stand belongs to the drug store hierarchically, and the traffic lights at the crossing belong to the street network hierarchically. However, when the traffic lights are red, people are more likely to look at the exhibited newspapers, and therefore its exact placement and value depends on the exact placement of the traffic lights. When trying to model this using the Split Grammar implementation, a natural approach would mirror the hierarchical structure, and would therefore suffer from exactly the same problem. Figure 2.8 illustrates this problem.

Another case where this limitation becomes noticeable is when modeling a building facade, and window sills are to be added. Usually, window sills protude from the wall, but this is hard to add retroactively, since the part that is outside the wall has already been split away in the beginning. To work around this, the wall has to be made thicker from the beginning to account for the potential addition of window sills in a completely different part of the grammar. This is contrary to how buildings are designed and therefore surprising and confusing for the user of the grammar system. In other scenarios, like when trying to align multiple grids, no such work-around is even possibly.

As can be seen in the paper introducing the existing implementation [18], the system allows to limit



(a) In a tree structure, separations between nodes emerge naturally **(b)** A semi-lattice structure allows distinct branches to join again

Figure 2.8: Tree vs. semi-lattice

the evaluation depth. However, when modeling complex structures, different sub-structures may require different depths of evaluation for a comparable level of detail. A better approach would therefore allow annotations to provide a semantic level-of-detail.

Chapter 3

Extending Shape Grammars with Context-Sensitive Sub-Rules

3.1 Approaching the solution

To overcome the limitations discussed in section 2.3.3, several approaches were tried and refined until the final solution was reached. Of course, since the grammars are actually implemented in a Turing complete programming language, any and all limitations can be overcome by scripting. However, for inverse modeling and automatic processing of the resulting shape description, the most restrictive system possible, that is still expressive enough for complex modeling tasks, is desired.

Starting point for approaching the solution was the existing split grammer implementation, to which additional features were added. This section describes the approaches that did not lead to the desired result, including their problems and limitations as well as the lessons that were learned by pursuing them and how the contributed to the idea of locally, context-sensitive shape grammars.

3.1.1 Approach 1: Using 3D Space as ”Storage Container”

The basic implementation placed convex polyhedra in 3D space as soon as a terminal-symbol was encountered during derivation. The first approach was to later collect these shapes again and replace them by a different one. This mirrors the context-sensitivity as used in formal grammars. For example, when placing several structures supported by columns on a landscape, it may be desireable to merge adjacent columns. This can be achieved by using a geometric query that defines what ”adjacent” means exactly, and collects columns into groups which can then be connected. In fact, for this task this approach works quite well, as can be seen in figure 3.1. The problem with this approach is that it does not scale well. The difficulty in defining geometric queries for different kinds of connections and especially evaluating grows exponentially with the model size, since all possible combinations have to be taken into account. Note that not even in architecture a purely geometric approach is used for connections, since all parts of a structure also have a ”purpose”, which could be interpreted as a label. It is therefore desireable to use this additional information to support context-sensitivity.

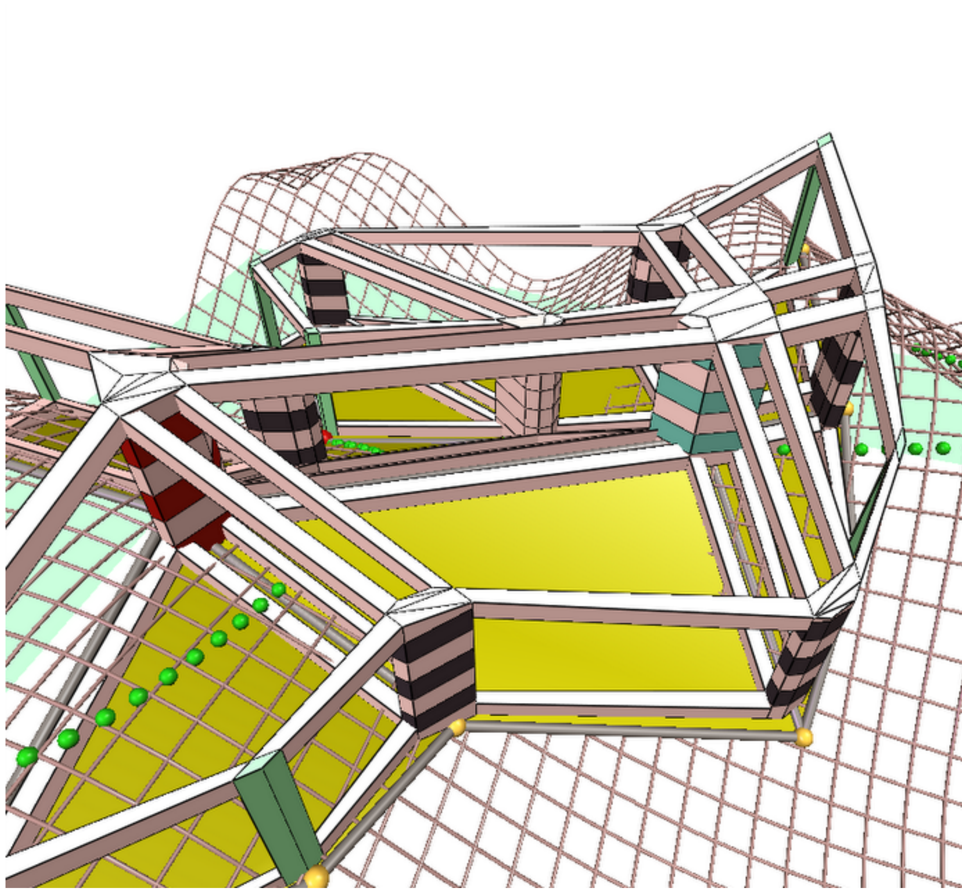


Figure 3.1: Several structures supported by columns are placed on quadrilaterals. Adjacent columns are merged into a single one using a simple geometric query to detect adjacence.

3.1.2 Approach 2: Defining and Using Attaching Points

To solve difficulty of defining geometric queries of approach 1, the idea devised by Krecklau et al [26], where each substructure declare where other structures may be attached as so called ”attaching points”, was taken as basis. In their work, different attaching points can be connected using connection patterns or even geometric queries, later on. The problem with this approach is that the connection process is completely decoupled from the grammatical description of the structures, therefore creating what is in essence a second system in addition to the grammar system. It is also not possible to model anything but true ”connections”, as discussed in section 2.1.2. However, the idea of ”attaching points” fits very well with the idea of named non-terminal symbols in a grammar. The problem of having a separate system for actually creating the connections still remained, however, as there was no integrated way for using the named non-terminals afterwards. What was needed was a way to manipulate the grammar tree retroactively, to select the appropriate nodes and perform operations on them.

3.1.3 Approach 3: Exchanging Information between Non-Terminal Symbols

The next idea was to devise a way to operate on the non-terminal symbols. This was achieved by enabling each non-terminal to register ”message handlers”, which were functions associated with a message

name. Other non-terminals could then send messages, tagged with a name, to specific non-terminals or broadcast them to all, and all registered message handlers that were associated with that name would be called. The theory was that this would allow non-terminal symbols to ”negotiate” where connections were made. Indeed, the approach allowed elegant solutions to some problems, like having a wall with windows, and placing a rainwater gutter between the windows. This was done placing the gutter at a fixed position and then sending a message to the windows to move themselves, as shown in figure 3.2. The reverse approach of having the windows send messages to the gutter to move has the problem that multiple messages, each adding its own constraint, would have to be taken into account. Restricting the number of constraints that are allowed per non-terminal symbol to one worked around this issue, but the resulting message handlers required a lot of custom code which was basically used to model the other constraints, which was also undesireable. Also, the way a non-terminal moved ”itself” was by sending a message to its parent requesting the split parameters to be changed, so if multiple siblings receive requests, only one of them could actually request a placement change from the parent, requiring even more code to ensure only requests that are really necessary are forwarded.

The message passing mechanism of this approach was then abandoned when it became clear that

(a) A raingutter is placed on a wall, and the windows move to avoid it. **(b)** If the raingutter is moved sufficiently, the result does not look natural anymore.

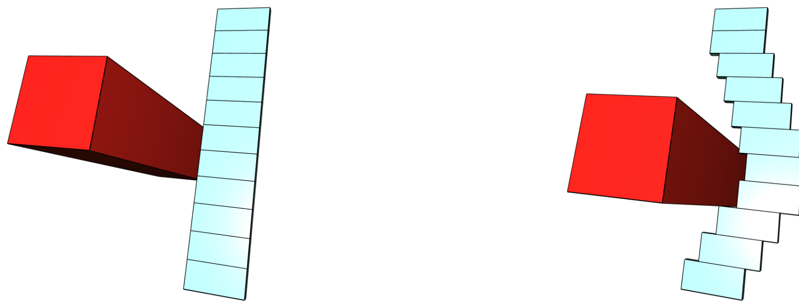
Figure 3.2: Message passing example

the message handler would just consist of complex code, which does not really fit with a clean, minimal formalism. However, the attempt gave important insight into how the grammar tree can be manipulated retroactively.

3.1.4 Approach 4: Using Optimization Algorithms on Split Parameters

A different way to look at the problem is to see the split parameters as variables within certain limits, which express the natural constraints of the construction. Further constraints modeling the interactions between sub-parts of the structure can then be added. Some of these may actually not even be ”hard” constraints, but just associated with a ”cost”, that should be minimized over the entirety of the constructed structure. This is exactly the kind of problem the field of mathematical optimization tries to solve, and several well-researched algorithms and approaches for it exist. Figure 3.3 shows how split parameters

were optimized such that a path avoids a column, while also minimizing the tension between the path elements. This approach works quite well for simple problems, where the constraints can be formulated easily, but does not scale well. Furthermore, some important issues are not addressed at all, like actually placing a physical connection between elements, and others, like aligning multiple grids are cumbersome to model.



(a) A straight path is placed next to a column

(b) The column is moved, so that it would intersect with the path. The split parameters of the path elements are optimized such that the tension between the elements is minized and the column avoided using a simple particle swarm optimization [22]

Figure 3.3: Optimizing split parameters

3.1.5 Towards Context-Sensitive Sub-Rules

All these attempts solved some, but not all sub-problems that were to be solved. However, by reexamining them, another, cleaner, yet also more powerful approach was found. To come to this conclusion, consider the grid alignment problem again (see figure 3.4).

The places where a grid is to be placed can be thought of as "attaching points", but instead of passing messages directly between these non-terminals, it would be more logical if their common ancestor performed the connection. After all, this is also how real structures are planned and constructed. So, instead of "storing" the non-terminal symbols in 3D space, their existence is made known to their ancestors, which can then decide which non-terminals to connect and how. By allowing this connection to be another application of grammar rules, the mechanism operates fully within the shape grammar formalism. The connection operation could also modify the local split parameters if necessary, alleviating the need for a complex optimization procedure.

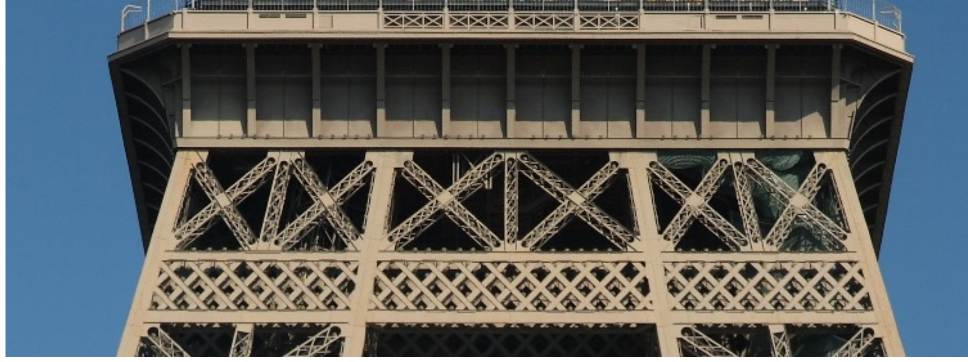


Figure 3.4: A photograph of part of the Eiffel Tower (Released under CC-BY-SA 3.0 by Benh Lieu Song) in which the support grid of one of the platforms of the Eiffel Tower is shown. Note how the grid bars align over the distinct grids.

3.2 Context-Sensitive Sub-Rules

With the realization described above, the split grammar mechanism can be extended by also allowing context-sensitive rules. However, if this was allowed at any point, it would be necessary to always search the whole scope tree for tuples matching the left hand-side of any context-sensitive rule, which is computationally expensive. It was therefore decided to restrict the context-sensitive rules to be local to another rule. Since it is possible to have sub-rules of the start-rule, this does not restrict expressibility in any way, but it allows finer-grained control for the user, and simpler evaluation. This also allows the user to more easily avoid name clashes. Also of note is, that, in contrast to context-sensitive grammars for languages, the context-sensitive sub-rules presented here do not take adjacence into account in any way. This means, that even two non-terminal symbols in completely different branches of the scope tree can be connected easily. The exact evaluation of these sub-rules is presented in section 3.2.2.

3.2.1 Definition

Given a rule $L \rightarrow R\,[fn_1]$, which will be called the "parent rule", another rule $A \rightarrow B\,[fn_2]$ can be added as "sub-rule". This is then written as

$$\begin{aligned} L \rightarrow & R\,[fn_1] \\ & \{A \rightarrow B\,[fn_2]\} \end{aligned}$$

As was discussed in section 2.1.1, the left hand-side of rules can usually only consists of a single non-terminal symbol. For the local sub-rules this restriction is lifted, so A can also consist of a list of non-terminal symbols $N_1\,N_2\,\ldots\,N_n$, and/or the special notation $[N_i]$, for some or all of the non-terminals N_i. Note, that unlike in context-free formal grammars, there are no restrictions placed on the right hand-side of rules.

Since the number of sub-rules per parent rule is restricted to one, constructs like this are often neces-

sary:

$$
\begin{aligned}
L \to & L_1 \,\text{[id]} \\
& \{C \to D\,[fn_3]\}
\end{aligned}
$$

$$
\begin{aligned}
L_1 \to & R\,[fn_1] \\
& \{A \to B\,[fn_2]\}
\end{aligned}
$$

Where id is the identity function, as shown below in section 3.3. Note that $A \to B$ will be evaluated before $C \to D$, as described in the next section. For the convenience of users, the same can also be written as:

$$
\begin{aligned}
L \to & R\,[fn_1] \\
& \{A \to B\,[fn_2]\} \\
& \{C \to D\,[fn_3]\}
\end{aligned}
$$

3.2.2 Evaluation of the Sub-Rules

The application of a particular rule is defined to be *finished* if each symbol on its right hand-side is either a terminal symbols or a non-terminal symbol and that non-terminal symbol has been *completed*. A particular instance of a non-terminal symbol is defined to be *completed* when the rule associated with it has been finished, and the sub-rule of that rule, if present, has also been finished. This means, a rule is *finished* only once the sub-tree corresponding to this rule has been constructed completely, including all sub-rules of referred non-terminals. Only in case a rule is finished, its corresponding sub-rule is evaluated. The scopes corresponding to non-terminal symbols on the right hand-side of a rule are added as children of the scope of the left hand-side of the rule. Together they form this rules local scope tree.

Then, for each N_i on the left hand-side of the sub-rule, a scope lookup in the local scope tree is performed, and (any) one of the scopes with that name is *selected*. For the special form $[N_i]$, **all** scopes with that name are selected. The grammar operation may then use any of those names to refer to the selected scopes and produce new scopes to which the labels listed on the right hand-side are then assigned and derivation on these new scopes continues as usual.

Of course, to be able to write useful rules using multiple scopes, new grammar operations have to be defined that can actually work with multiple input scopes. A proposed set of grammar operations is described in section 3.3, including a description how this set can be extended easily.

3.3 Extended Grammar Operations

As described above, additional grammar operations are required, which can handle multiple input scopes as arguments. The result of each of those operations can then again be multiple scopes. For maximum flexibility, each of these operations also accepts an list of scopes where a single scope is expected, unless

otherwise noted. How this list is handled depends on the operation, but care was taken to make the result intuitive. For example, an operation that usually operates on a single scope and returns a single scope, will just perform its operation on each scope in the list and return the list of result scopes.

3.3.1 List of Grammar Operations

Identity Operation

In cases where scopes should be given a "new" name, the identity operation is sometimes useful. This is especially true if a substructure shall be used twice, but given two different names. This is possible by introducing an additional layer. It is also necessary for rewriting rules that would require additional sub-rules, as demonstrated above. The identify operation takes a single argument, and returns a single result scope, which has exactly the same properties as the argument scope. For a list of input scopes, the identity operation is performed on each of them, and a list of the results returned.

Radial Split Operation

The original version of split grammars only allowed interval splits by parallel translation of a single plane. However, to construct arches or similar structures, it is desirable to have an operation that produces "round" structures. The radial split allows the construction of such structures. It takes a single scope, two planes, a point in space and an interval as its arguments . The split moves the two planes such that they both contain the point, and then performs a spherical linear interpolation, as described in [38], between the two normal vectors. The values in the parameter vector are used as the angle steps between the two normals, with negative values used as relative steps, as for the straight split (see section 2.3). If the two planes are parallel, a third plane must be given, which will be used as "midpoint", as to disambiguate the arc that is described by the normal vectors.

Constructive Solid Geometry

The constructive solid geometry (CSG) operations "union", "intersection" and "difference" are provided as grammar operations. Each takes two scopes and returns a list of scopes, corresponding to a convex decomposition of the resulting geometry. This is necessary, since the result of a CSG operation on two convex polyhedra is not necessarily convex itself. When operating on lists of scopes, the operations behave as if the list was joined into a single (not necessarily convex) geometry with "union" and the operation applied afterwards.

Extruding Scopes

An operation that has been found to be useful (see section 5.2.4), is the ability to increase the size of the geometry associated with a scope. This is done in two different ways:

- Extend the geometry infinitely in one direction
- Extend the geometry in all directions for some amount

Extending infinitely is useful when looking for overlaps in a particular direction, for example to connect different parts with nails or rivets. Extending for a fixed amount can be used to attach a connector to an already existing geometry. Both operations work on lists by extending each scope in the list independently and returning a list of all result scopes.

Convex Hull Operation

This function takes a single input scope or a list of input scopes as argument and performs the convex hull operation on them. It then returns a single scope with a geometry corresponding to that convex hull.

Collecting Child Scopes Surrounded by a Contour

When multiple convex shapes are put together, holes may appear between them. It is often useful to find those holes and apply further rules to them. This is actually a two-part problem, since first the holes have to be found, and since they are not necessarily convex, they have to be decomposed into convex parts afterwards. An alternative use for the contour is to "decorate" it by putting scopes in equal distance along the contour.

Find Contour Operation This operation takes a list of scopes and a plane, and returns a list of contours these scopes describe on the given plane. There will be one contour of the outline and one for each hole, where the outline is stored as a list of edges in clockwise order, the holes as lists of edges in counter-clockwise order, so that the "outside" is always on the left of the edge. These contours can only be used for the two operations "convexify" and "decorate".

Convexify Operation This operation takes a contour, which is always a simple polygon, and returns a convex decomposition of that contour, which can then be filled with convex polyhedra, creating new scopes.

Decorate Operation This operation takes a contour, a distance and a step length and returns a list of scopes, which are aligned along the contour with a distance of the given step length, and extending to the given distance towards the "outside" (left side of the contour edge).

3.3.2 Variable Number of Result Scopes

For some grammar operations it is not possible to predict the number of result scopes, especially those involving CSG operations on complex geometries. Since the right hand-side of each rule must consist of the same number of non-terminal symbols as the number of result-scopes returned by the operation, this would make writing correct rules for these cases impossible. To allow the writing of rules for these cases, the keyword "repeat" is introduced:

$$L \rightarrow R \, [fn_1]$$
$$\{A \rightarrow \text{repeat:} B \, [fn_2]\}$$

The repeat-keyword states that the rule B is to be applied to all result scopes returned by fn_2.

3.3.3 Combining Grammar Operations

Since all grammar operations that operate on scopes also take lists of scopes as arguments, and return lists of scopes, it is desirable to use the return value of one of these operations as the argument of another. In practice, this allows the construction of almost arbitrarily complex grammar operations as compositions of simpler ones. In fact, this type of composition is already possible given the above definition. For example, consider that something like this is desired:

$$
\begin{aligned}
L \rightarrow & R\ S\ T\ [\text{split-interval}([-1,-1,-1],X)] \\
& \{R\ S\ T \rightarrow \text{repeat:}B\ [\text{union}(\text{diff}(R,S),\text{diff}(T,S))]\}
\end{aligned}
$$

The same operation can be expressed like this:

$$
\begin{aligned}
L \rightarrow & R, S, T\ [\text{split-interval}([-1,-1,-1],X)] \\
& \{R\ S \rightarrow \text{repeat:}X\ [\text{diff}(R,S)]\} \\
& \{S\ T \rightarrow \text{repeat:}Y\ [\text{diff}(S,T)]\} \\
& \{[X]\ [Y] \rightarrow \text{repeat:}B\ [\text{union}(X,Y)]\}
\end{aligned}
$$

It is therefore merely convenience for the user to allow the direct composition of grammar operations, and not an extension of the definition.

3.3.4 Extending the Set of Grammar Operations

The given set of grammar operations can be extended almost arbitrarily. If care is taken that all new grammar operations accept both scopes and list of scopes, and return either single scopes or lists of scopes, they can be easily integrated into the existing definition and composed arbitrarily with the existing grammar operations. Similiar to the split-operation it is also possible to expect arguments of data types other than scopes.

3.4 Scope Parameters

In many situations (see section 5.2.1 for examples), many rules that are almost the same, with the exception of some (usually numeric) parameter or set of parameters, are required. Of course, it is possible to simple instantiate a new rule, including possibly changed sub-rules, for each parameter-value, for example by having `rule-1`, `rule-4`, `rule-5` for the parameter-values 1, 4 and 5, but this gets tedious when big and/or numerous rules are involved. To improve this, parameters were introduced.

3.4.1 Defining Scope Parameters

Each rule may define arbitrarily named parameters and assign them values. Each rule may then refer to any of its parameter by name on the right hand-side. This is written by putting all parameter assignments above the rule where they are defined.

$$
\begin{aligned}
& [p_1 = v_1, \ldots p_n = v_n] \\
& \qquad\qquad L \rightarrow R\ [fn(p_1, p_n)]
\end{aligned}
$$

3.4.2 Scope Parameter Inheritance

With just the ability to define parameters local to rules, the problem of similar rules has not been solved. It must also be possible to define parameters for rules from outside, such that different references to these rules result in them having different values for the parameters. This is solved by inheriting parameter values to all rules referenced on the right hand-side. When a parameter is referenced, then, the name is looked up recursively in the referring rules until one matching the name is found. To use a rule A with specific parameter values it is therefore only necessary to set those parameters in the referring rule, and then have A on the right hand-side. Another rule in the grammar can then set different parameter values, and also use A on the right hand-side.

3.4.3 Scope-Local Coordinate Systems

One set of parameters that deserves special treatment is the coordinate system. To faciliate rule re-use of structures, it is often useful to give them a local coordinate system. This way, when a structure is used multiple times in different rotations, the rules can still refer to X-, Y- and Z-direction in the usual manner. To make the inheritance and setting of local coordinate systems, the following coordinate system options can be used:

- `use-X`, `use-Y`, `use-Z`: Use the given direction
- `use-split`: Use the direction in which the split was performed
- `cross-X-split`, `cross-split-X`, `cross-Y-split`, ...: Perform a cross product of the normals of the X-/Y-/Z-direction and the split direction, in the given order, and use the result as the normal for the new direction
- `auto`: Derive one coordinate direction from the other two

Each option can be used for each of the three coordinate directions X, Y and Z. For example:

$$L \rightarrow R, S, T \text{ [split}(X, X = \text{use-split}, Y = \text{use-Z}, Z = \text{auto})]$$

This would set the new X-direction in the direction of the split (in this case the X-direction of the parent scope), the Y-direction in the Z-direction of the parent-scope and calculate the Z-direction based on the X- and Y-directions. Only one coordinate direction can be declared as `auto`, and the options involving the split-direction are only available for split-operations.

3.4.4 Inheriting from Multiple Scopes

Local, context-sensitive sub-rules add another dimension to the inheritance question, since now scopes could have multiple parents. To avoid the complexity that comes with this, the newly added scopes are actually derived from the parent scope of the outer rule, which makes inheritance simpler. The only exception to this rule is the coordinate system, which is inherited in an operation-defined way, as to enable the coordinate system options described above. Even in cases where no split is performed, this makes the behavior more intuitive, since a connection attached to some substructure would be expected to share the coordinate system of that substructure.

3.4.5 Semantic Level-of-Detail

To demonstrate how this parameter-concept can be used and extended, support for level-of-detail evaluation has been added. It works by using a parameter called `lod-level` that should be defined in the start-rule with the desired level of detail. The special operator lod can then be used to provide two alternatives for the next level of detail. For example:

$$L \rightarrow \text{lod}((R, S, T\ [\text{split}(X)]), (F))$$

When this rule is encountered during evaluation, the current value of the parameter `lod-level` is consulted. If it is greater than 0, the first parameter is used as the right hand-side of the rule. Otherwise, the second parameter is used. The intent behind this is to give the user the ability to define ”logical” level of detail-steps. For example usage of this see section 5.2.12

3.5 Nondeterminism in the Evaluation of the Grammar

As described in section 3.2.2, the local, context-sensitive rules will match ”one” scope with the corresponding label. If there are multiple scopes with the same label, any of them is a possible match, introducing non-determinism in the evaluation of the grammar. If a deterministic evaluation is wanted, the user has to rename scopes accordingly to be able to uniquely identify which scopes to select.

3.6 Substructure Interfaces using Labeled Non-Terminal Symbols

A big advantage of this approach is that sub-structures can be reused and also exchanged with different ones. For this to work, a notion of the ”interface” a substructure provides has to be formed. There are two directions of ”information flow” in the grammar, up-wards and down-wards the scope-hierarchy. The down-wards information flow consists of the parameter values, so each substructure has to declare which parameters it requires to be present. The up-wards information flow consists of the scope names, which are selected by parent-scopes in their local, context-sensitive rules. For substructures to be interchangeable, they have to expose all the labels the parent requires. Therefore, it is important to declare which labels are provided by a substructure. By analysis of the grammar, it can be deduced which labels are required by all parent rules, and only if the set of provided labels is a superset of the set of required labels, the substructure can be inserted. See section 5.2.2 for an actual interface description.

Chapter 4

Implementation

4.1 Extensions to the Split Grammar Implementation

For the implementation of the extensions described in chapter 3, several modifications and extensions to the existing implementation of split grammars were necessary. Some of these changes are only incidentally related to the local, context-sensitive subrules, and could also be useful in a purely context-free split grammar implementation. They are discussed in this section.

4.1.1 Radial Split

The radial split operator discussed in 3.3.1 was implemented as a GML-function. It takes four parameters, two planes, S and E, a vertex v, and an interval list I_i. In case the two planes are parallel, a fifth parameter, another plane, M, that should be in the middle of the two planes along the desired arc, is required. The function then proceeds as follows:

1. Store the normals n_S and n_E of the planes S and E.
2. Determine the angle Ω between the normals n_S and n_E.
3. If the angle is 0 or 180: Store the normal of M as n_M
4. Else: Store the angle bisector of n_E and n_S as n_M
5. Initialize s_p and s_n with 0
6. For each interval I_i do:
   (a) If I_i is positive: Add I_i to s_p
   (b) Else: Add abs(I_i) to s_n
7. Set r to $\Omega - s_p$. This determines how much of the angle can be used for relative sizes
8. Set n to r/s_n. This determines how much one "unit" of relative size is in absolute terms.
9. Store 0 as the current angle α_c

10. Set n_A to n_S and n_B to n_M
11. Set Ω_0 to the angle between n_A and n_B. This is the angle between the start and the midpoint.
12. For each interval I_i do:
    (a) If I_i is positive: Set i to I_i
    (b) Else: Set i to $-I_i \cdot n$, to convert it to an absolute size
    (c) Add i to α_c
    (d) If α_c is greater than Ω_0: Set n_A to n_M and n_B to n_E, and subtract Ω_0 from α_c. This marks the passing of the mid-point.
    (e) Perform a spherical linear interpolation between n_A and n_B, to get the normal vector n_c at the angle α_c.
    (f) Emit the plane with the current normal vector n_c as normal vector, and place it in the vertex v

The list of emitted planes is then used to split the given scope into subscopes, in the same way as for parallel splits. For example:

```
(0,0,0) (50,10,20) 5 scope-box
[ 20 -1 90 -1 50 ]
CPX CPZ CPZ cp-pl-flip (25,0,0)
split-radial
F V F V F
```

Here, a box of width 50 is split into 5 parts. The split is an arc between the local Z-plane and its flipped version, therefore encompassing an angle of 180 degrees. Since the arc is not unique, the X-plane is also given as mid-point. Also, the split is performed by rotation around the point $(25, 0, 0)$. Of the five pieces, the first has an angular size of 20 degrees, the third of 90 degrees and the fifth a size of 50 degrees. The remaining two pieces each get half of the remaining angle, in this case $(180-90-50-20)/2 = 20/2 = 10$ degrees. Figure 4.1 shows the result of the split, when the first, third and fifth piece are filled and the other two are empty.

4.1.2 Scope-Local Coordinate Systems

For local coordinate systems it is often convenient to allow the automatic setting of coordinate directions in the direction of the performed split. To allow this, and the other coordinate system settings described in section 3.4.3, special variants of the split functions, denoted by an appended `-c` are provided. They take an additional first parameter, which has to be an array of three functions that are executed to set the X-, Y- and Z-direction, respectively. For example

```
    { cs-use-split cs-use-Z cs-auto }
    [ -1 -1 -1 ]
    CPY
```

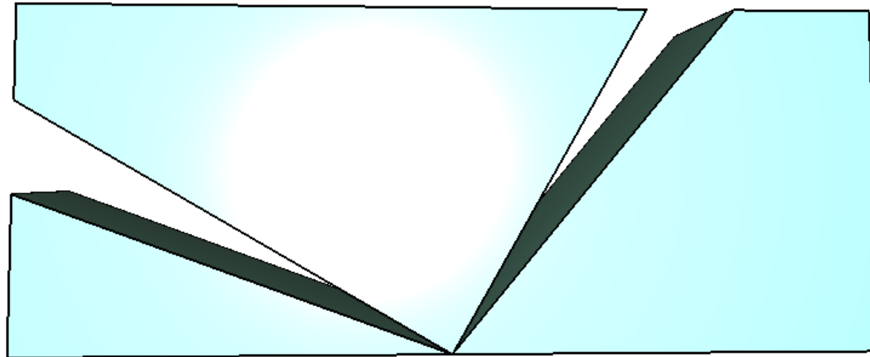


Figure 4.1: Result of a radial split

```
4      split−interval−c
5      R S R
```

This would perform a split in Y-direction, and set up the new coordinate systems of the sub-scopes such that their local X-direction is the direction of the split, their local Y-direction is the Z-direction of the parent and their local Z-direction is such that it is orthogonal to both their local X- and Y-direction.

4.2 Context-Sensitive Sub-Rules

4.2.1 Defining Rule Context

As described in section 2.3.1, in the split grammar implementation each rule was represented by a function in GML, and the correspondence to grammar rules is only incidental. Basically, a scope stack was maintained, and all functions operated on the top of that stack. A split-operation consumed the top element of the stack and pushed new ones corresponding to the newly created scopes, which were then consumed in their rules in turn. To implement local, context-sensitive rules, a context in form of the parent rule is required after all operations and further derivation has occured. Furthermore, this can happen on multiple levels in the scope tree. A pair of functions `begin−rule` and `end−rule` is provided that handles opening and closing the current context. A typical rule may look like this:

```
1  /L {
2      begin−rule
3      [ −1 −1 −1 ]
4      CPX
5      split−interval
6      R S R
7      end−rule
8  } def
```

The rules R and S will also be enclosed in `begin-rule`/`end-rule`-pairs. The scope to which a `begin-rule`/`end-rule`-pair refers is called the **rule-scope**.

`begin-rule` is implemented by referring to the top element of the scope stack, as before, and using it as the beginning of a GML scope. `end-rule`, on the other hand, simply closes that GML scope again. Since scopes were implemented as dictionaries in the original implementation (but note section 4.4.2), this enables access to all values defined in a scope even after it has been consumed from the scope-stack. This is utilized by the local, context-sensitive rules to access the subscopes of the scope where the rule is defined.

4.2.2 Setting Scope Labels

Since there is no real notion of "current function", and since it is entirely possible to have "anonymous" scopes in the sense that they don't have a corresponding GML function, the labels for the scopes have to be set explicitely. For this, the function `set-name` is provided, which takes a label from the stack and sets the name of the rule-scope to that label. In addition to simply closing the rule-scope, `end-rule` has the additional task of informing the parent-scope of all labels defined in the rule-scope. This is done by maintaining a dictionary, mapping from names to lists of scopes, which contains all labels defined in sub-scopes. `end-rule` then simply adds all entries from the rule-scope's dictionary to its parent's dictionary, appending to an existing list where necessary.

4.2.3 Implementing Lazy Evaluation of the Grammar

In the original implementation, whenever a `terminal-fill`/`F` was encountered, the current scope's convex polyhedron was immediately placed in the scene. With local sub-rules it can now happen, that these polyhedra are supposed to be replaced later on, which would require to remove them from the scene again. While this is certainly possible, it is much more efficient to just store a flag that states that a polyhedron is to be rendered, and clear it again in case it is replaced by a subrule. Then, only after the grammar has been evaluated, all polyhedra that have the flag set, are actually rendered. To perform this final pass, the function `create` has to be called with the top level scope as parameter.

In fact, even more information is conserved in the scope tree. Each scope remembers each split operation that was performed on it, and its children, including children that are the result of subrules. Therefore, after the grammar evaluation, the scope tree stores almost all the information that would be necessary to re-evaluate the grammar. It would therefore be possible to retroactively change any split parameter if desired. This is not strictly necessary for context-sensitivy, but see section 6.3 for an outlook on what this could be used for.

4.2.4 Scope Selection and Sub-Rules

To select scopes for use in sub-rules, the functions `get-scope` and `get-scopes` are provided. As described in section 4.2.2, a dictionary of defined labels is maintained in each scope. When selecting a particular scope for a subrule, the name is simply looked up in the rule-scope's dictionary, returning a list

of scopes. The function `get-scope` simply provides the first value from that list, whereas the function `get-scopes` provides the whole list, which corresponds to the []-special form in the grammar rules, as described in section 3.2.2.

4.3 Grammar Operations

All grammar operations are implemented as GML-functions taking their corresponding arguments from the stack. As mentioned in section 3.3, both lists of scopes as well as just scopes themselves are accepted as arguments where appropriate. This means, that both `get-scope` as well as `get-scopes` can be used to select the parameters. For example:

```
1    /A get-scope
2    /B get-scopes
3    connect-intersection
4    make-scopes
```

Will produce the intersection between the first scope in the list of scopes labeled A, and all scopes labeled B. This result can then be used for further grammar operations. To finalize the result, which includes pushing the resulting scopes on the scope stack, the function `make-scopes` is used. Afterwards, arbitrary rules can be applied to the produced scopes. As described in section 3.3.2, the number of result scopes may not always be known, and indeed in this very example, it depends on how many B-scopes exist and where they are located relative to the selected A-scope. `make-scopes` also provides the system with the necessary information to support the `repeat`-keyword. In GML, it is written in postfix-notation like any other looping construct and named `repeat-rule`, for example:

```
1    /A get-scope
2    /B get-scopes
3    connect-intersection
4    make-scopes
5    { C } repeat-rule
```

This would apply the rule C to all scopes produced by the intersection.

4.3.1 List of Added Grammar Operations

- `connect-convex`
  - **Parameters:** `scope-list`
  - **Description:** Results in one scope consisting of the convex hull covering all scopes in the given list
  - **Sample code:**

```
/A {
  begin-rule
  /A set-name
  LF
  end-rule
} def

/B {
  begin-rule
  /B set-name
  LF
  end-rule
} def

/S {
begin-rule
[ -1 -1 -1 ]
CPX
split-interval
A V B

[
/A get-symbol
/B get-symbol
]
connect-convex
make-scopes
LF
end-rule
} def
```

– The **result** can be seen in figure 4.2.

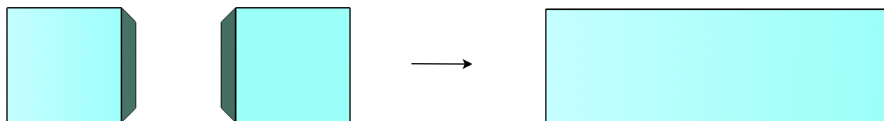


Figure 4.2: Result of the `connect-convex` grammar operation

- `connect-union`

  – **Parameters:** `scope-A scope-B`

- **Description:** Results in a number scopes corresponding to the union of the two given scopes. Both parameters may also be lists, but care must be taken that the scopes in each list are pairwise disjoint prior to the operation, since this is required by the underlying GML-operator. This operation is basically only useful to join multiple scopes into a list for further processing.
- **Sample code:**

```
/A {
  begin-rule
  /A set-name
  LF
  end-rule
} def

/B {
  begin-rule
  /B set-name
  LF
  end-rule
} def

/S {
begin-rule
[ -1 -1 -1 ]
CPX
split-interval
A V B

/A get-symbol
/B get-symbol
connect-union
make-scopes
{ LF } repeat-rule
end-rule
} def
```

- The **result** can be seen in figure 4.3.

• `connect-difference`

- **Parameters:** `scope-A scope-B`
- **Description:** Results in a number scopes corresponding to the difference of the two given scopes, that is $A - B$. Both parameters may also be lists, but care must be taken that the

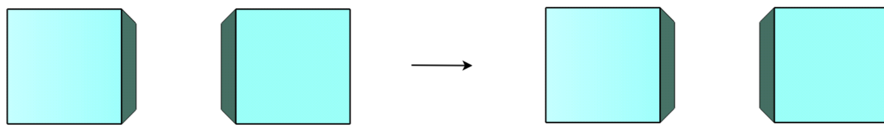


Figure 4.3: Result of the `connect-union` grammar operation, as can be seen, the result is exactly the same as the input. The only difference is that all input scopes are put together into one list of scopes.

scopes in each list are pairwise disjoint prior to the operation, since this is required by the underlying GML-operator.

– **Sample code:**

```
/A {
  begin-rule
  [ -2 -1 -2 ]
  CPZ
  split-interval
  LV A1 LV
  end-rule
} def

/A1 {
  begin-rule
  /A1 set-name
  LF
  end-rule
} def

/B {
   begin-rule
   /B set-name
   LV
   end-rule
} def


/S {
begin-rule
[ -1 -2 -1 -2 -1 ]
CPX
split-interval
A V B V A

/A1 get-symbols
```

```
33 connect-convex
34
35 /B get-symbols
36 connect-difference
37 make-scopes
38 { LF } repeat-rule
39 end-rule
40 } def
```

The **result** of this code can be seen in figure

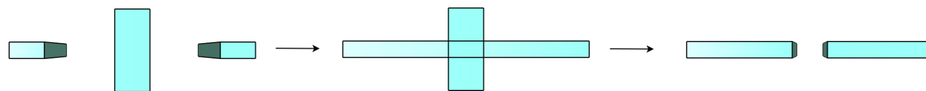


Figure 4.4: Result of the `connect-difference` grammar operation. First, an intersection has to be created, which is not possible using only splits. Therefore, the two scopes named `A1` are connected using `connect-convex`, and the difference of this convex hull minus the scope named `B` is used.

- `connect-intersection`
  - **Parameters:** `scope-A scope-B`
  - **Description:** Results in a number scopes corresponding to the intersection of the two given scopes. Both parameters may also be lists, but care must be taken that the scopes in each list are pairwise disjoint prior to the operation, since this is required by the underlying GML-operator.
  - **Sample code:**

```
1  /A {
2    begin-rule
3    [ -2 -1 -2 ]
4    CPZ
5    split-interval
6    LV A1 LV
7    end-rule
8  } def
9
10 /A1 {
11   begin-rule
12   /A1 set-name
```

```
13     LF
14     end−rule
15   } def
16
17   /B {
18      begin−rule
19      /B set−name
20      LV
21      end−rule
22   } def
23
24   /S {
25     begin−rule
26     [ −1 −2 −1 −2 −1 ]
27     CPX
28     split−interval
29     A V B V A
30
31     /A1 get−symbols
32     connect−convex
33
34     /B get−symbols
35     connect−intersection
36     make−scopes
37     { LF } repeat−rule
38     end−rule
39   } def
```

The **result** of this code can be seen in figure 4.5.

Figure 4.5: Result of the `connect-intersection` grammar operation. First, an actual intersection has to be created, which is not possible using only splits. Therefore, the two scopes named `A1` are connected using `connect-convex`, and the intersection of this convex hull with the scope named `B` is used.

- `extend-infinite`
  - **Parameters:** `scope direction`
  - **Description:** Results in a scope that is the same as the given scope, except that all planes parallel to the given direction are removed, which in effect extends the scope to infinity in

the given direction, if such a plane was present. If a list of scopes is given, the operation is performed on each one, and the list is a result of the resulting scopes.

– **Sample code:**

```
/A {
  begin−rule
  /A set−name
  LV
  end−rule
} def

/S {
  begin−rule
  id
  A

  /A get−symbol
  /PZ extend−infinite
  make−scopes
  { LF } repeat−rule
  end−rule
} def
```

The **result** of this code can be seen in figure 4.6.

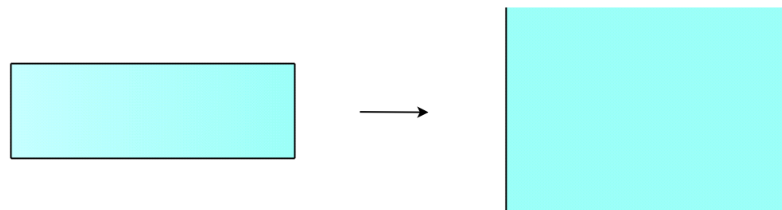


Figure 4.6: Result of the `extend-infinite` grammar operation

- `extend-scope`

  – **Parameters** `scope distance`

  – **Description** Results in a scope that is the same as the given scope, with all planes moved outwards by the given distance. If a list of scopes is given, the operation is performed on each one, and the list is a result of the resulting scopes.

  – **Sample code:**

```
/A {
  begin−rule
  /A set−name
```

```
4    LF
5    end-rule
6  } def
7
8  /S {
9    begin-rule
10   id
11   A
12
13   /A get-symbol
14   3 extend-scope
15   make-scopes
16   { LF } repeat-rule
17   end-rule
18 } def
```

The **result** of this code can be seen in figure 4.7.

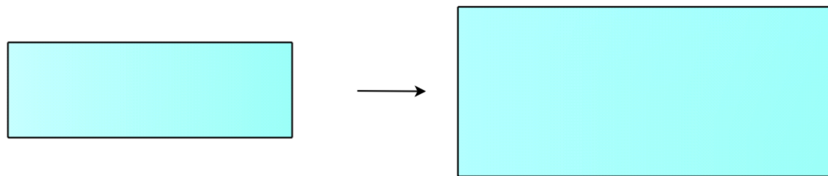


Figure 4.7: Result of the `extend-scope` grammar operation

- `connect-rod`
  - **Parameters** `scope-A scope-B dimensions`
  - **Description** This function serves as a demonstration on how the set of grammar operations can easily be extended for special purposes. It takes two scopes, and a 2D vector specifying the size, and connects them with a rectangular hexahedron, where the local coordinate system is set such that the x-, y- and z-directions are parallel to the sides of the rectangular hexahedron. The length is given by the start and end scopes, while width and height along that length are given by the 2D vector.
  - **Sample code:**

```
1  /A {
2    begin-rule
3    [ -2 -1 -2 ]
4    CPZ
5    split-interval
6    LV A1 LV
7    end-rule
```

```
8   } def
9
10  /A1 {
11    begin−rule
12    /A1 set−name
13    LF
14    end−rule
15  } def
16
17  /S {
18    begin−rule
19    [ −1 −4 −1 ]
20    CPX
21    split−interval
22    A V A
23
24    /A1 get−symbols aload
25
26    (1,1) connect−rod
27    make−scopes
28    LF
29
30    end−rule
31  } def
```

The **result** of this code can be seen in figure 4.8.

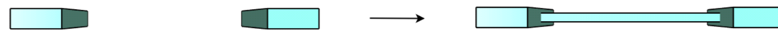


Figure 4.8: Result of the `connect-rod` grammar operation

Collecting Child Scopes Surrounded by a Contour

Polygon Representation Since planes offer more significant bits than vertices, as described in section 2.2.4, polygons are not represented as a list of vertices, but rather as a list of planes, together with a base plane. This allows the representation of polygons with vertices that might otherwise not be representable exactly. Each vertex of the polygon is then represented by the intersection of two adjacent planes in the plane list and the base plane. By convention, the list of planes is stored in clockwise direction when going around the polygon. This allows the representation of ”holes” in the same way, by storing them in counter-clockwise direction. This way, when walking along the edge of the polygon in storage order, the ”free space” is always on the left.

Find Contour The algorithm is given a list of convex polyhedra $c_{(i)}$ and a baseplane b. It first determines all line segments that can be part of the contours, and then generates all contour-polygons on the baseplane by joining the appropriate line segments. To determine the line segments $l_{(i)}$ the algorithm proceeds as follows:

1. Add b to all c_i
2. For each c_i do
   (a) Determine the planes p_j of c_i adjacent to b in clockwise order
   (b) For each p_j generate a line segment l, where the line is the intersection of p_j and b and with start- and end-planes p_{j-1} and p_{j+1}, wrapping around as necessary
   (c) Add l to the generated line segments $l_{(i)}$
3. For each l_i do
   (a) For all other l_j do
      i. Remove all parts where l_i and l_j coincide from l_i

The last loop is necessary because the polyhedra may have common planes, which are then not part of the outline. However, not all parts of these planes may coincide, so only parts have to be removed. Figure 4.9 shows an example. With all these line segments the contours $cn_{(i)}$ can now be generated as follows:

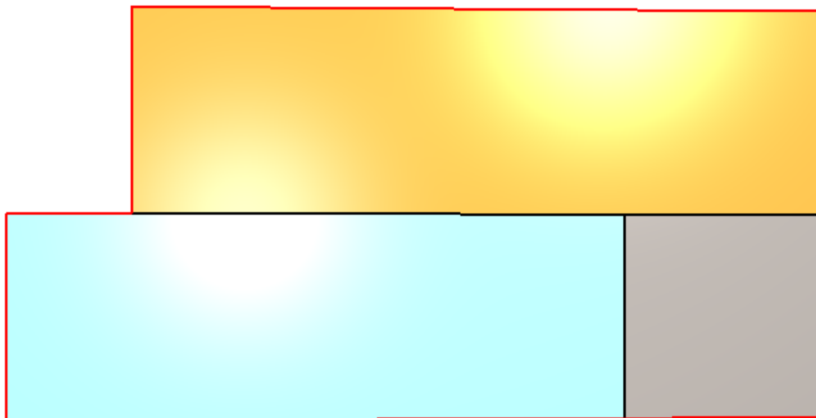


Figure 4.9: In this example, the cyan and the gray polyhedron share some planes, but since they are fully aligned, no part of these edges will be part of the contour, and so they can be fully removed. However, the shared planes of the cyan and the yellow polyhedron do not fully "cancel out", so only a part of that edge may be removed. The resulting contour is shown in red.

1. While there still is a line segment $l_{(i)}$ do
   (a) Choose any line segment l_i

(b) Start a new contour cn_i with l_i as its only line

(c) Store the start point of l_i as start point s

(d) Store the end point of l_i as current point e

(e) While $s \neq e$ do

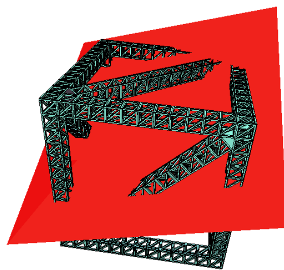
i. Find all line segments $o_{(j)}$ with start point at current point e

ii. Determine the next line segment (see below) o_j

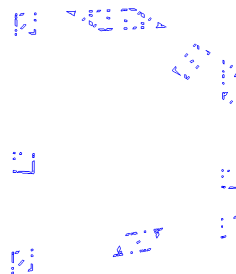
iii. Add o_j to the current contour cn_i

iv. Set the end point of o_j as the current point e

In most cases, this algorithm works straightforward, care must only be taken with vertices with valence greater than 2, that is, when multiple polyhedra touch in a single point on the baseplane b. In this case, there are really only two sensible approaches: Either the contour encompasses all polyhedra touching in such a point, or only one. The latter approach leads to further complications when the contours are then used subsequently, because then multiple contours will coincide in a single point, therefore the first option was chosen. To do that, all line segments starting at the intersection points are sorted clockwise by angle from the current line segment, and the first one of those is used. Figure 4.11 shows an example case.



(a) A plane is put through a complex structure, and ”find contour” is invoked.



(b) The result of the ”find contour” operation is a list of polygons, corresponding to the outline of the convex polyhedra on the given plane.

Figure 4.10: ”Find contour” example

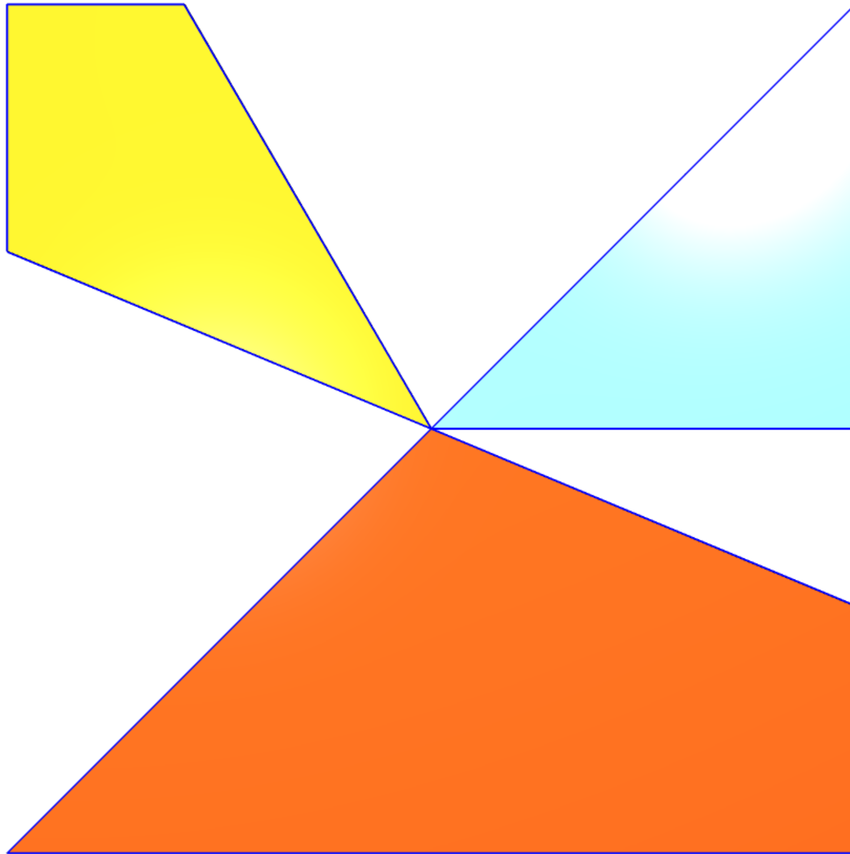


Figure 4.11: In this example, the polyhedrons all meet in a single point. The two sensible choices are to have either one contour for each polyhedron, or a single contour that encompasses all. The latter was chosen because it makes further processing easier. Note that in practice this scenario is rare, since even if several polyhedrons appear to share a common vertex, they may not, due to numerical issues. In such a case, there will be several contour-polygons, but since the vertices are not exactly the same, this is not a problem for further processing.

Convexify Different algorithms exist for the convex decomposition of polygons, depending on what properties the decomposition is required to have [20], [8]. For the purpose of this implementation, the focus was put on exactness, using the integer coordinates of the existing implementation. Since there is no guarantee that any additional planes can even be represented by integer coordinates, only already existing planes are used. The algorithm is given a polygon as a list of planes p_i, plus a base plane b, and performs the convex decomposition as follows:

1. For each plane p_i do:
    (a) Consider the vertex v described by p_i, p_{i+1} and b (wrapping around to p_0, if necessary)
    (b) If v is a reflex vertex, calculate the intersections c_j of p_i with the polygon
    (c) Of all intersections c_j choose the one as d that is closest to v, and towards inside the polygon as seen from v
    (d) Split the polygon along the line $\overline{vd}$ into parts A and B
    (e) Recursively convexify the parts A and B and return the union of the results.
2. If no reflex vertex is found, return the polygon itself.

4.4 Scope Parameters

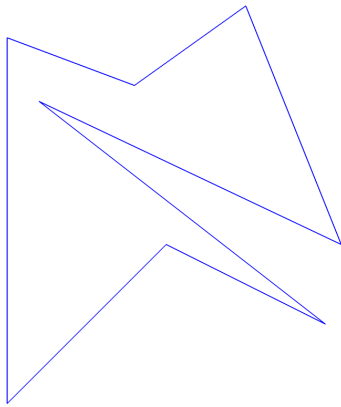
4.4.1 Setting and Getting Scope Parameter Values

Since each `begin-rule` results in the current scope being pushed onto the dictionary stack, with `end-rule` popping it again, any `def` that occurs in that immediate scope results in the value being added to the current scope. This means, setting parameters for the current scope works exactly like assigning variable values in GML, and any name is looked up in that scope. For example:

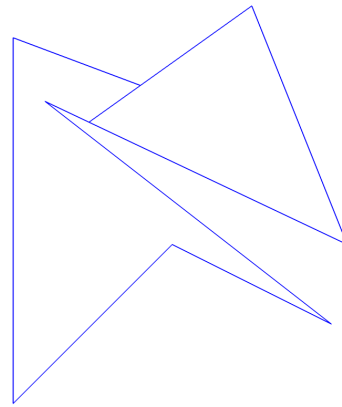
```
1  begin-rule
2  /width 10 def
3     [ -1 -1 width ]
4     CPX
5     split-interval
6     R S R
7  end-rule
```

4.4.2 Scope Parameter Inheritance

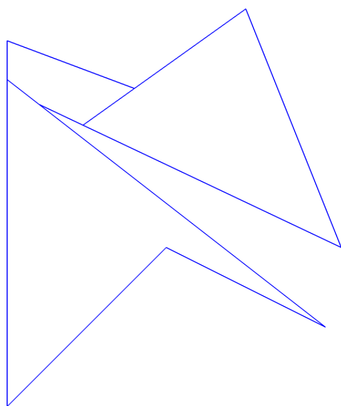
With the above mechanism, it is actually already possible to use the parameter values in all sub-scopes as well, as long as no scope is used outside its parents' function. However, local, context-sensitive rules allow scopes to be selected anywhere above them in the hierarchy. To preserve the dictionary hierarchy, each scope therefore needs to remember its parent scopes values as well. This is done by using dictionary stacks rather than normal dictionaries as scopes. A dictionary stack is, as the name implies, an ordered



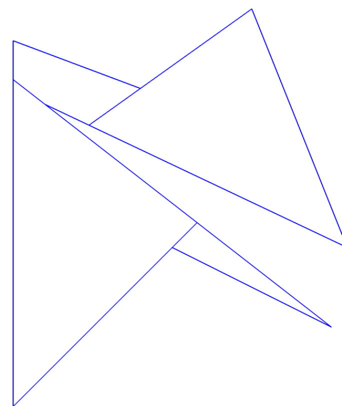
(a) A non-convex polygon, for which a convex decomposition shall be computed



(b) A reflex vertex is found, and the first plane in clockwise direction is extended towards the inside of the polygon until it hits the opposite side. The polygon is then split into two parts, and the convex decomposition repeated on both parts



(c) One of the parts is already convex, so it is used as-is, while in the other, another reflex vertex is found, and again the first plane in clockwise direction is extended, and the convex decomposition repeated on both parts



(d) Another reflex vertex is found, a plane extended, and then all parts are convex, so the algorithm terminates

Figure 4.12: ”Convexify” example

collection of dictionaries, with one of them designated as the ”top” of the stack. When a name is looked up, it is first searched for in the top dictionary, and if it is not found, in the next one below that. Each scope now consists of a copy of its parent’s dictionary stack, to which it adds a new dictionary to the top. Since the dictionaries themselves are only referenced and not copied, each scope sees changes in its parent’s dictionary in its own dictionary stack.

Chapter 5

Example Models

5.1 Simple Examples

5.1.1 Grid Tiles with Connections

One of the simplest examples that demonstrates what can be done consists of several tiles arranged in a grid. If it is then desired to connect several - potentially different - of these tiles, the limits of context-free Shape Grammars are reached quite quickly. Of course, it is possible to model this, by taking the connections into account from the beginning, but a more logical approach would be to model the tiles and their connections independently from each other. The approach described in Chapter 3 provides exactly these capabilities. As long as tiles fulfill a set interface, they can look any way that is desired, and can then still be interconnected, by using a local, context-sensitive rule. For example, if there is a grid cell that exposes four non-terminal symbols `northeastcorner`, `northwestcorner`, `southeastcorner` and `southwestcorner` as non-terminal symbols, and it is desired to connect two adjacent grid cells in some fashion. For this, first two cells have to be placed somewhere, for example like this:

```
1  begin—rule
2  id
3  cell
4  /southeastcorner /southeastcorner1 rename—symbol
5  /southwestcorner /southwestcorner1 rename—symbol
6  end—rule
7
8  begin—rule
9  id
10 cell
11 /northwestcorner /northwestcorner2 rename—symbol
12 /northeastcorner /northeastcorner2 rename—symbol
13 end—rule
```

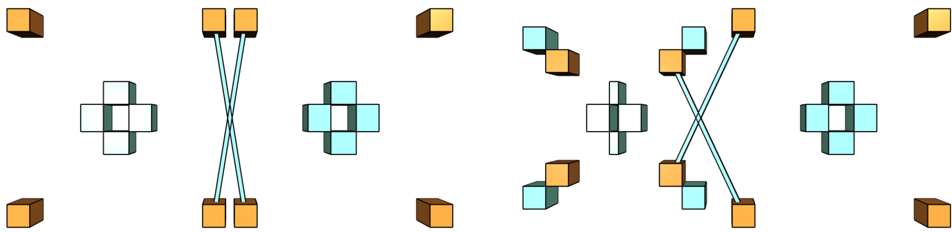
In this case, the symbols for the corners had to be renamed so they can be uniquely identified later. This is equivalent to having local subrules using the identity function, as described in 3.3.1, but the

implementation also allows this short-hand notation. After that, the corners of the two cells can be connected:

```
/northwestcorner2 get−symbol
/southeastcorner1 get−symbol
(0.02 ,0.02)
connect−rod
make−scopes
LF

/northeastcorner2 get−symbol
/southwestcorner1 get−symbol
(0.02 ,0.02)
connect−rod
make−scopes
LF
```

The result of this calls can be seen in figure 5.1.



(a) Two simple grid cells, with the corner connectors in orange, are connected

(b) The code for the connections does not change at all, even if one (or both) cell is replaced, as long as it has the same interface, that is it exposes the required non-terminal symbols

Figure 5.1: Cell grid example

5.1.2 Wooden Planks with Nails

Another problem that is simple to solve with local subrules, but hard to do without context-sensitivity consists of two layers of planks above each other, which are to be connected where they overlap. A simple example for this are wooden planks used in a fence where nails are to be put where the different layers overlap. Solving this with the grammar functions presented above works as follows: The two plank layers are expected to expose the non-terminal symbols `plankx` and `planky` respectively, which

contain all planks in the layer. First, all planks are extended infinitely in the direction of the overlap. Then the intersection of these extended scopes is calculated. This intersection, is still infinitely long, is the area where the nails have to be placed, but they should be limited to just the planks, plus some protusion. To do this, first the original planks are extended with an offset equal to the desired protusion, and then the infinitely long scopes are intersected with these newly extended planks, limiting the room where the nails are placed to the overlap of the planks, extended by some offset in the overlap direction. Then, the nails simply have be placed in the center of each of these scopes. The code to do this is:

```
1       /plankx get-symbols
2       /PZ extend-infinite
3 
4       /planky get-symbols
5       /PZ extend-infinite
6     connect-intersection-d
7 
8         /plankx get-symbols
9         /planky get-symbols
10      connect-union
11    0.1 extend-scope
12  connect-intersection-d
13  make-scopes
14  { make-nail } repeat-rule
```

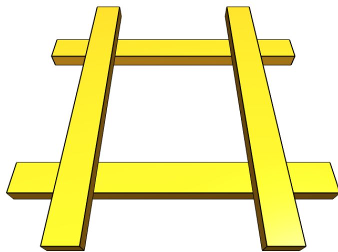
Noteworthy is the use of the `connect-intersection-d` grammar function, since the `-d`-suffix disables the previously placed planks, which would render them invisible. This is necessary because placing the nails would create overlapping geometry, which is usually undesirable. So, in the next step, the holes for the nails are made by using the `connect-difference` grammar function.

```
1     /plankx get-symbols
2     /planky get-symbols
3   connect-union-d
4   /nail get-symbols
5   connect-difference
6   make-scopes
7   { LF } repeat-rule
```

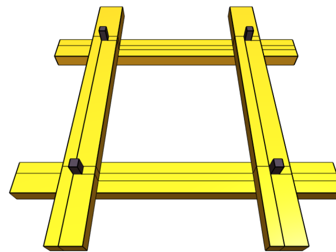
The result of this code can be seen in figure 5.2

5.1.3 Grid Spanning Multiple Separate Areas

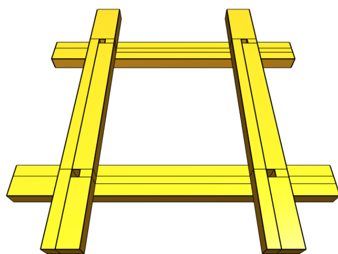
For aesthetic reasons, it is often desired to align the bars of multiple different grids that appear in a structure. In a normal, context-free grammar all the grids would be generated independently, resulting in misaligned bars, since it is not possible to take other parts of the model into account. The CGA Shape implementation allows to solve this problem by defining ”snap lines”, which the bars of all grids can subsequently use for alignment [33]. But another way to look at the problem is that this is really



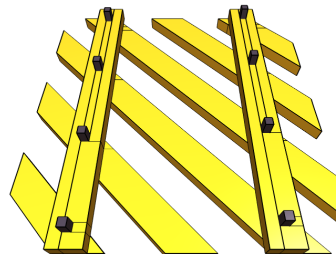
(a) Two layers of planks are to be connected with nails



(b) Nails are placed in the overlapping parts



(c) If `connect-difference-d` is used to make the holes for the nails, the nails themselves also become disabled, showing the holes



(d) As long as the plank layers fulfill the same interface, any configuration of planks can be connected using the same code

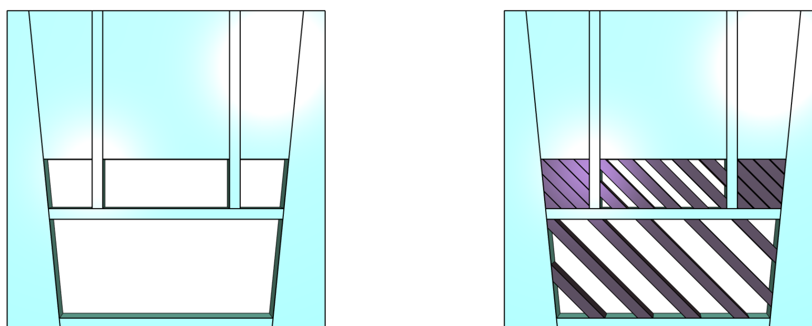
Figure 5.2: Planks connected with nails

a "connection" between the different grids, which can be solved by local, context-sensitive rules. For example:

```
1  begin-rule
2  id
3  grid-containers
4
5  /grid-here get-symbols connect-convex
6  make-scopes
7  make-grid
8
9  /grid-here get-symbols
10 /grid-beam get-symbols
11 connect-intersection
```

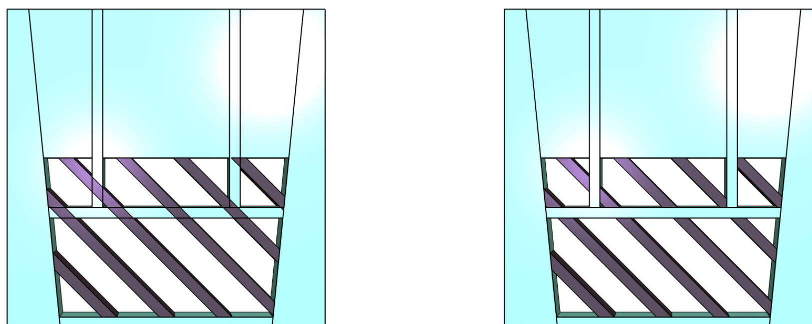
```
12    make-scopes
13    { LF } repeat-rule
14    end-rule
```

Here, first a rule `grid-containers` is called, which is expected to have non-terminal symbols called `grid-here` wherever a grid shall be placed. Then, a local rule takes all these places and connects them using a convex hull, and places a grid in this convex hull. Since that means that the grid would also cross the boundaries between the different places where it should actually be placed, it needs to be cut again. The grid exposes its bars as non-terminals called `grid-beam`, which are then intersected with the `grid-here`-parts, to fit the grid exactly where it should be. Figure 5.3 shows an example.



(a) Each empty area should be fit with a grid

(b) A naive approach leads to unsatisfying results, since different holes require different grids. Even then, aligning the grids manually is tiresome and error-prone



(c) When the grid is fit into the convex hull, it overlaps the parting structure

(d) By calculating the intersection with the single `grid-here`-parts, the grids are restricted to where they should be

Figure 5.3: Multiple aligned grids

5.1.4 Contour Example - Convexify

Similiar to the tile example from above, a number of shapes is arranged in a grid. The difference is that now the holes are to be connected. The tricky part is that the holes are not convex, so the usual functions do not work. However, by combining the `find-contour` and `convexify`-operations it is possible to fill the interior of each hole with convex polyhedra, and use those polyhedra for further processing. One possibility is, like in the previous example, to fill the holes with a single grid that is aligned over all holes. Assuming the grid tiles expose the parts enclosing the holes as non-terminals called `rod`, this would be done like this:

```
1    /rod get-symbols
2    PZ
3    find-contour
4  convexify
5  make-scopes
6  { make-block } repeat-rule
```

After this, all holes would be partitioned into convex parts, each being its own scope and on each scope the `make-block` rule is applied. For the purpose of this example, it is assumed the `make-block` rule simply fills the scope with a single non-terminal called `block`. To get a grid aligned over all holes, the procedure is exactly the same as in the previous example. First, the blocks are connected using a convex hull, then a grid is placed there, then all `grid-bars` are intersected with the blocks to only leave them in the holes:

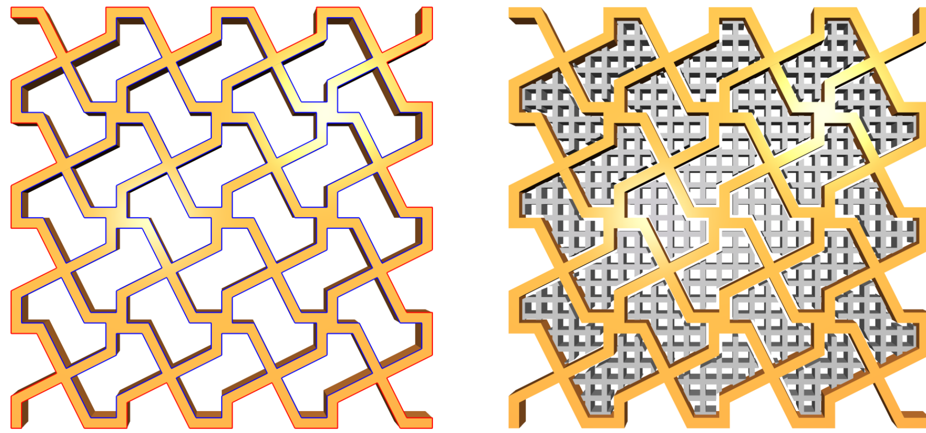
```
1    /block get-symbols
2    connect-convex
3    make-scopes
4    make-grid
5
6    /grid-bar get-symbols
7    /block get-symbols
8  connect-intersection
9  make-scopes
10 { LF } repeat-rule
```

The result of this operation is shown in figure 5.4

5.1.5 Contour Example - Decorate

As an alternative to filling the holes with some structure, it might also be desireable to have some sort of ”decoration” along the outline. The `decorate` grammar function does exactly that. For example:

```
1    /rod get-symbols
2    PZ
3    find-contour
4    (0.1 ,0.1 ,0.1) 0.075 0.125
```



(a) An arrangement of turnstile grid cells, the holes are outlined in blue

(b) A grid aligned over all holes

Figure 5.4: Fitting an aligned grid into holes between turnstile grid cells

```
5 decorate
6 make-scopes
7 { make-cylinder } repeat-rule
```

This will create an axis-aligned box scope with a size of $0.1 \times 0.1 \times 0.1$ every 0.125 units along the contour, with an offset of 0.075 towards the inside of the hole. It is then possible to call produce arbitrary objects at that position, for example cylinders as shown in figure 5.5.

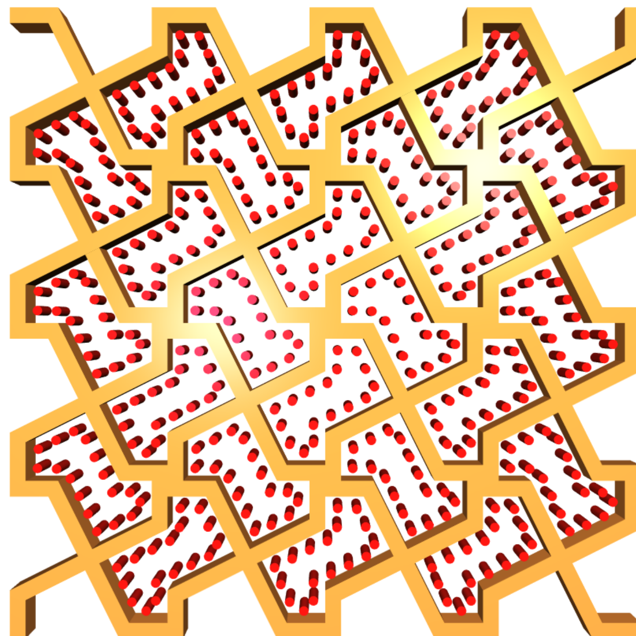


Figure 5.5: Decorating a contour outline

5.2 Eiffel Tower

5.2.1 General Structure

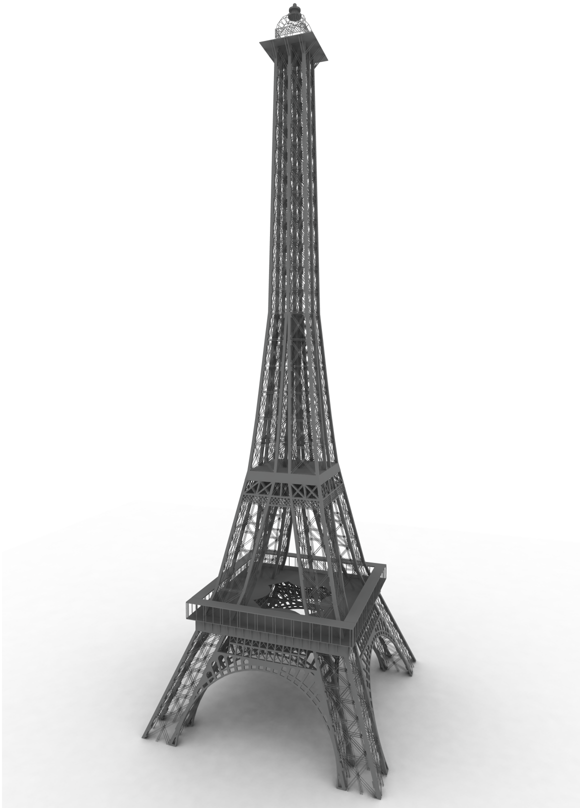


Figure 5.6: The Eiffel Tower model

When starting the model of a complex structure, it pays off to analyze which parts of the structure are similiar to each other and may therefore be described using a common rule-set. Especially noteworthy for such considerations are natural symmetries. In case of the Eiffel Tower, the square structure with 4 identical quadrants leads to a natural dissection into four parts. Each of those four parts consists of one leg with their lattice structure. To simplify the grammatical description, the legs were separated into their low, middle, higher and top parts, with the two platforms between the lower and middle, and the middle and higher parts, respectively. Each of the parts of a leg has the same basic structure of four strong support beams with a lattice of supporting struts between them.

To demonstrate the advantage of having local, context-sensitive rules, several connections that appear naturally in the Eiffel Tower were modeled in the Shape Grammar. For example, between the bottom parts of adjacent legs, arches were placed. Above that, the lower platform spans all four legs, and is supported by the arches. The middle part is reinforced by an additional grid connection right below the higher platform. Furthermore, the lattice structure contains diagonal and vertical bars that are connected where they overlap. Where they don't overlap, additional supporting struts are placed. The bars are also connected to the main beams of the legs. More subtly, the campanile features a dome that is mounted on a square cage.

An important resource when creating the model was the book "The 300 Meter Tower" by Gustave Eiffel, which contains many blueprints of all important parts [16].

5.2.2 Single Leg Interface

As already discussed in 3.6, the non-terminals exposed by a substructure together with the parameters it requires form the interface of that substructure. Figure 5.7 shows a single leg of the Eiffel Tower together

Figure 5.7: The interface of a single leg of the Eiffel Tower: The non-terminals shown in red are connected with the neighbouring leg and fit with the arches. All the non-terminals shown in cyan are connected to form the bottom platform. The orange parts are used for the upper platform support structure. Finally, the green part on the very top is used to only have a single lattice where two legs meet. All these connections are discussed in detail below.

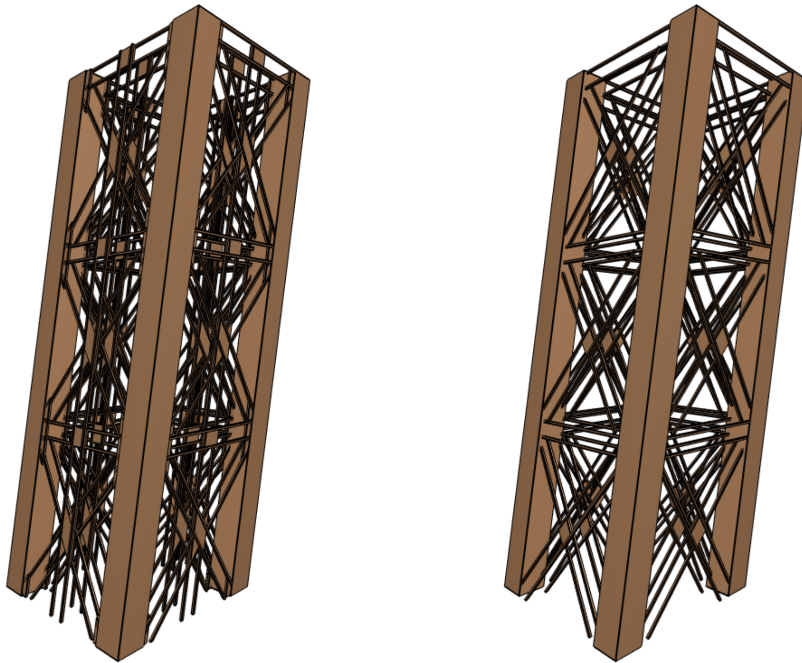
with the relevant non-terminals that are used for further processing. The second part of the interface is what parameters the subsubstructure requires, of which a short summary is given in table 5.1. Of course, other parameterizations, with their own advantages and disadvantages, are also possible. For example, one problem with this parameterization is that there are dependencies between the different parameters, especially the slopes and offsets. However, it is possible to calculate the slopes from the offsets when setting the values.

5.2.3 Support Beams with Lattice Structure

The principal component of the Eiffel Tower is a section of the main support beams with the accompanying lattice structure. There are two basic variants of this, one with and the other without a vertical bar in the middle. Figure 5.8 shows a segment of this part.

Name	Description
`lo-height`, `mi-height`, ...	Relative heights of the single parts of the leg
`lo-slope`, `mi-slope`, ...	Slopes of the single parts of the leg
`lo-offset`, `mi-offset`, ...	Offsets from the center for the parts

Table 5.1: Summary of required parameters for the leg



(a) A part of the main support beams with lattice structure. The lattice consists of vertical, horizontal and diagonal bars, which are connected where they overlap

(b) A simpler part of the main support beams with lattice structure. Here, the lattice is missing the vertical bar. This structure is used in the upper parts of the Eiffel Tower.

Figure 5.8: Main support beams with lattice structure

5.2.4 Connecting Bars with Rivets

Then, to provide more details, the connection is fitted with rivets, or in this case, simply holes where the rivets could then be placed, as seen in figure 5.9. This was achieved like described in section 5.1.2, just on a much larger scale.

5.2.5 Placing Arches

The cavities below the bottom parts of adjacent legs, shown in red in figure 5.7 above, are connected, and an arch is placed between them. The arches themselves make heavy use of the radial split operation. Figure 5.10 shows a detailed view of an arch.

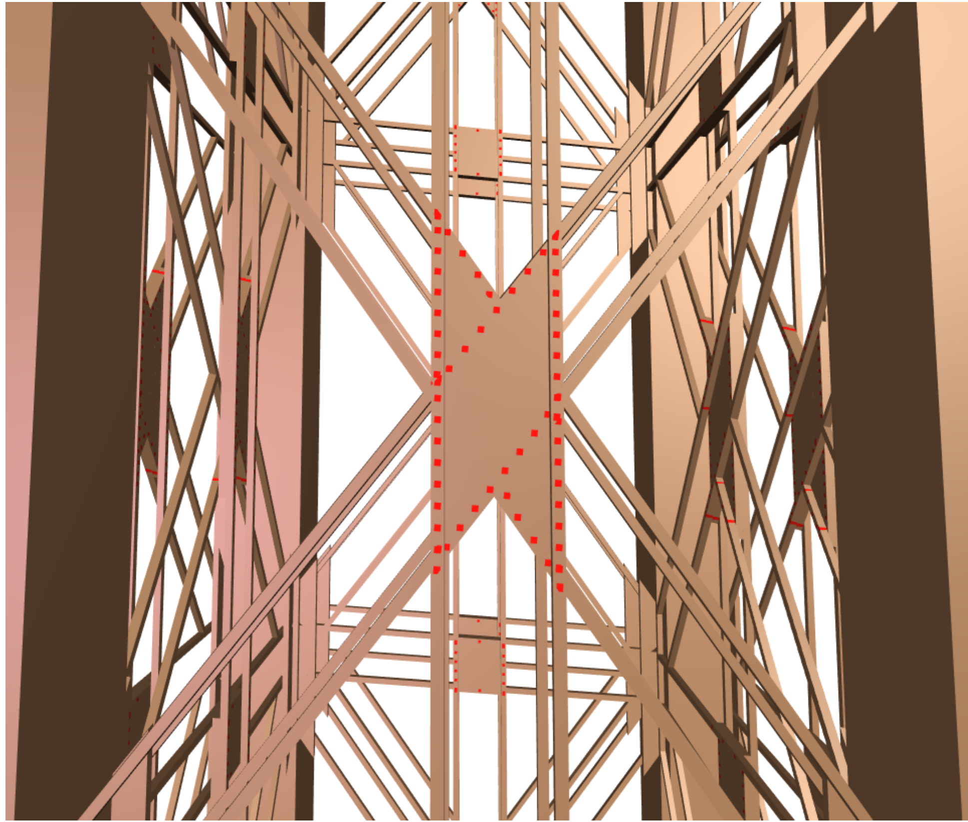


Figure 5.9: Holes for rivets are put where two bars overlap and have a connector.

Figure 5.10: One of the decorative arches of the Eiffel Tower, also note the lower platform, and the grid that connects the arches with the lower platform

5.2.6 Lower Platform

The lower platform consists of multiple parts. First, the floor is simply fit in all the relevant cavities of the lower platform part of the legs. Furthermore, a railing is mounted on the outside of each pair of adjacent legs, with additional supports below the floor. The corner parts of this railing does not necessarily have

to be in a local sub-rule, but as in software development, it has been proven useful to keep related things together. To support the floor, an additional grid is set up right below it, which connects the lower platform with the arches, which are, as described above, themselves connections between different legs, creating connections between connections. Figure 5.11 shows the lower platform railing. Below, in section 5.2.10 an alternative view from below the tower is shown, where the support grid and the hole in the platform can also be seen.

Figure 5.11: The lower platform of the Eiffel Tower

5.2.7 Higher Platform Support Structure

The higher platform is supported by additional grid structures right below it. Those grid structures connect each pair of adjacent legs in two places. In contrast to the arches, which are placed strictly between the legs, these grids also affect the legs themselves. This was achieved in the same way as any other connection, by including the legs on the left hand side of the local sub-rule, similiar to the grid example in section 5.1.3. Figure 5.12 shows how the smaller lattices are integrated between the legs, and how a single grid is fit along the whole width.

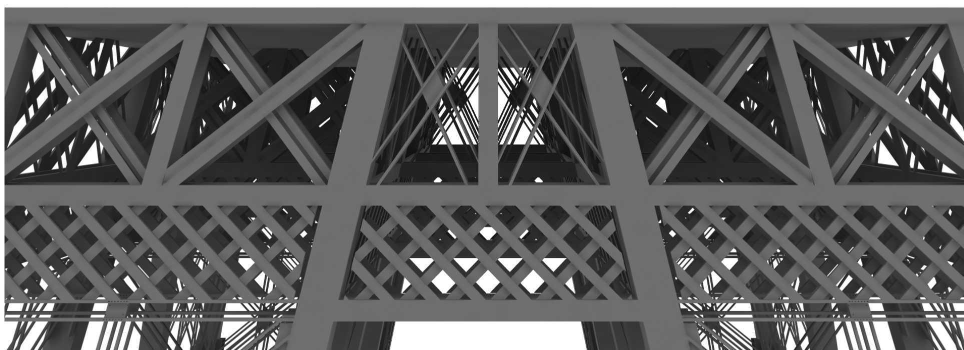


Figure 5.12: The supporting structure for the higher platform. Note how the grid is aligned over the whole width, and how the central lattice mirrors the structure of the other lattices.

5.2.8 Common Grid in Top Part

In the very top part of the Eiffel Tower, the single legs meet. If every leg contained a lattice on all four sides there, there would be two lattices right next to each other in the center. In reality, though, the legs ”join”, and only a single lattice is placed between them. This is easily modeled using a local sub-rule, as shown in figure 5.13

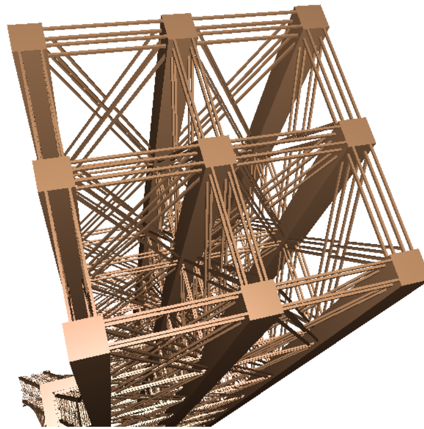


Figure 5.13: Shared grid between the top parts of the Eiffel Tower

5.2.9 Campanile

The campanile on top of the Eiffel Tower basically consists of several applications of the radial split, forming a dome structure, on top of which a round structure is mounted. Below the dome, however, is a grid structure on which the dome had to be mounted. This was done by creating a local sub-rule that connects the dome to the grid twice: One connection is responsible for having the support beams outside the grid structure, the other provides the horizontal support bars. Figure 5.14 shows the result.

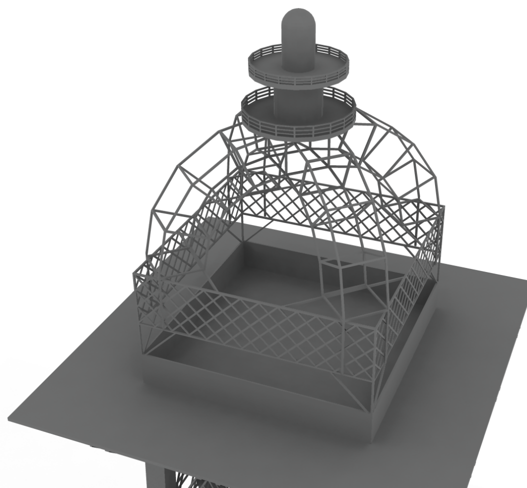
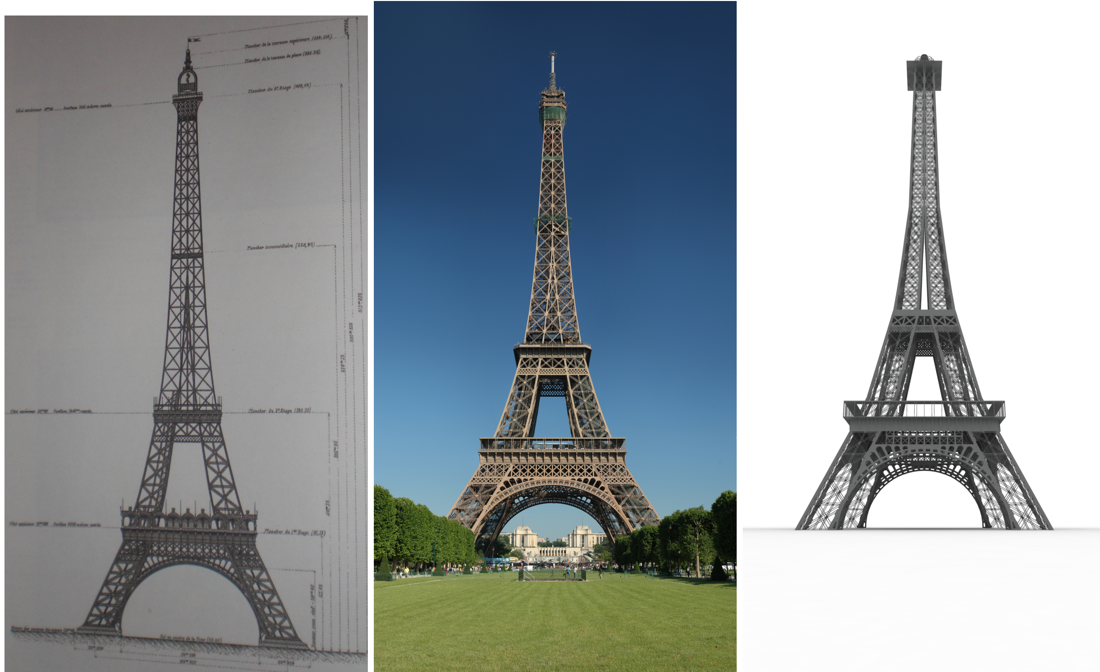


Figure 5.14: The campanile on top of the Eiffel Tower

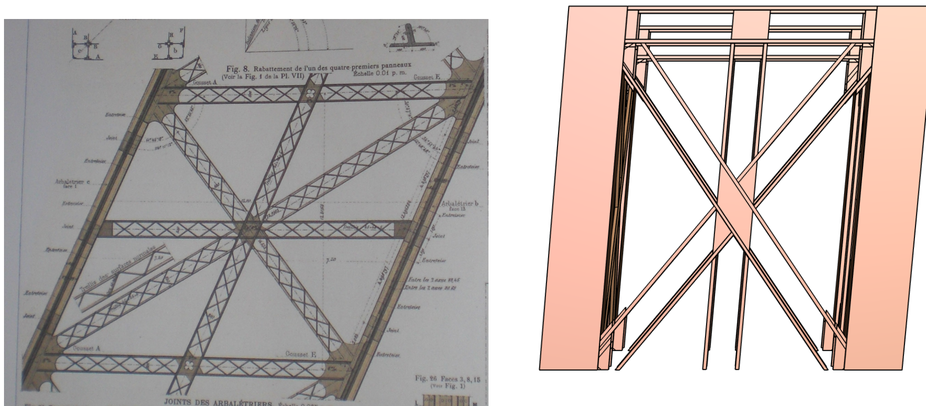
5.2.10 Evaluation of the Model

As mentioned above, the blue prints provided in the book "The 300 Meter Tower" [16] were used as basis of the model. To prove the quality of the model, several comparisons between the blue prints and photos of the actual Eiffel Tower and renderings of the created model are shown here.



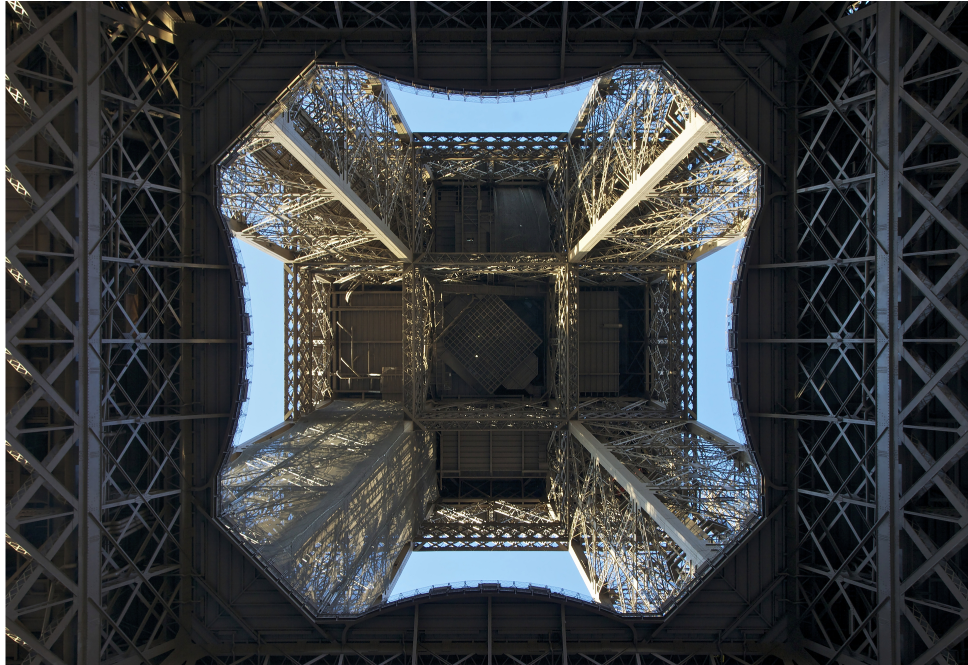
(a) The general structural blue print of the Eiffel Tower **(b)** A photograph of the Eiffel Tower (Released under CC-BY-SA 3.0 by Benh Lieu Song) **(c)** A rendering of the Eiffel Tower model

Figure 5.15: Comparison of the Eiffel Tower model with a blue print and a photograph

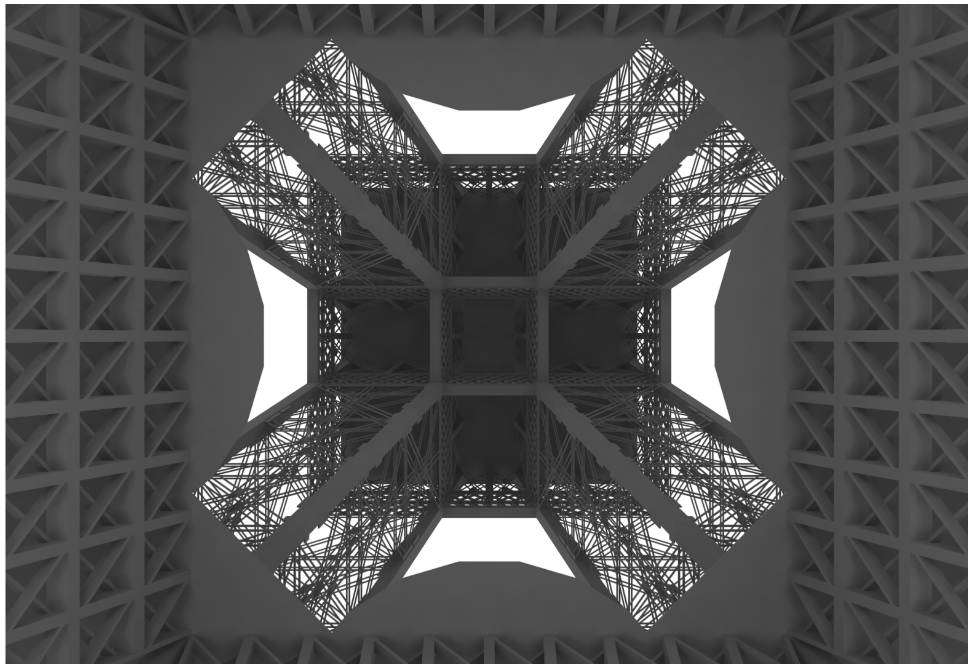


(a) A blue print of one section of the pillars **(b)** A detailed view of one of the pillars

Figure 5.16: Comparison of the Eiffel Tower model with a blue prints of a pillar part



(a) A photograph taken from below the Eiffel Tower (Released into the public domain under CC0)



(b) A rendering of the Eiffel Tower model

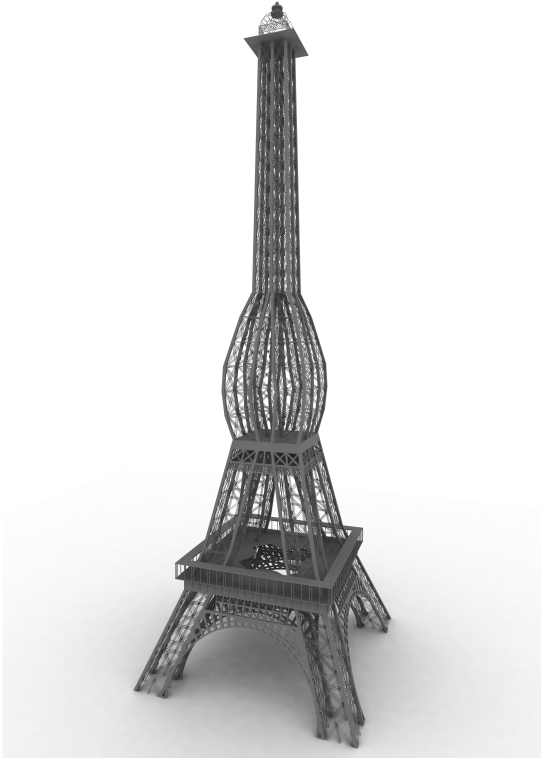
Figure 5.17: Comparison of the Eiffel Tower model with a photograph from below. Note the characteristic shape of the hole in the bottom platform

5.2.11 Eiffel Tower Variations

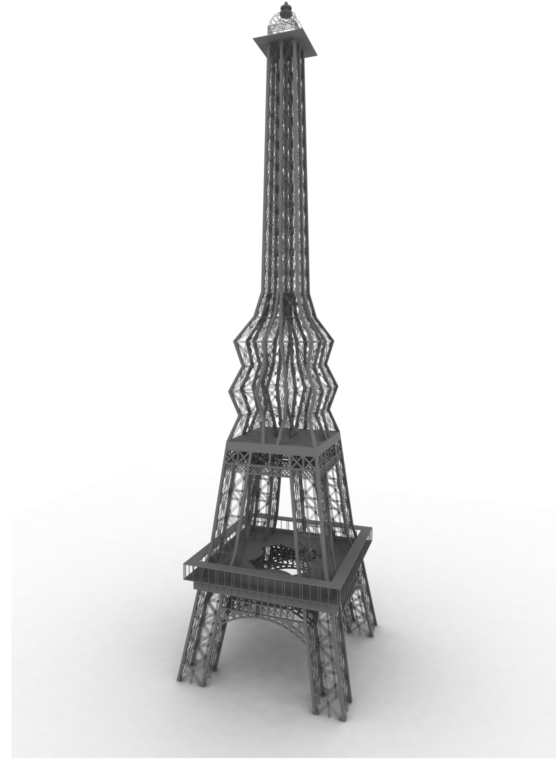
One of the reasons for a procedural model is the ease with which variations of the same model can be created. The parameterization of the model determines the design space one has when creating such variations. Since the Eiffel Tower model is defined as four pillars with fixed parts, but variable slopes and heights for these parts, variations along these dimensions are very easy. Figure 5.18 shows several altered Eiffel Tower models. Since the connections are also created procedurally, they stay consistent even when the model is changed.

5.2.12 Level-of-Detail

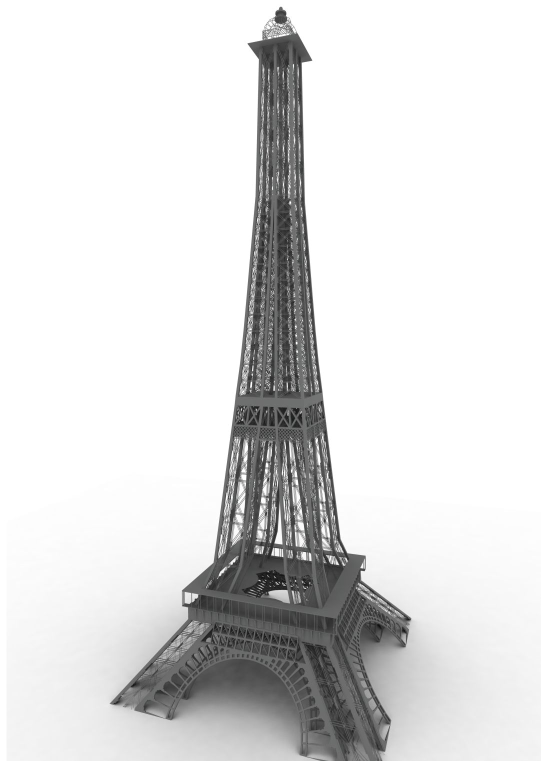
The description of the Eiffel Tower knows four level-of-detail levels. The coarsest version only consists of the general structure, with four legs, and no lattices, platforms, arches or any other detail. The second level adds a basic version of the campanile, platforms and arches, as well as the lattice structures to the legs, but the bars in the lattice are not split into single rails and there are no connections from them to the general structure. The next level then adds all connections between bars and adds all missing detail to the campanile, platforms and arches. The highest level, finally, only adds rivets to the connections. Since putting holes into convex polyhedra increases the number of required polyhedra considerably, this leads to a massively increased rendering time. Figure 5.19 shows the different levels of detail next to each other.



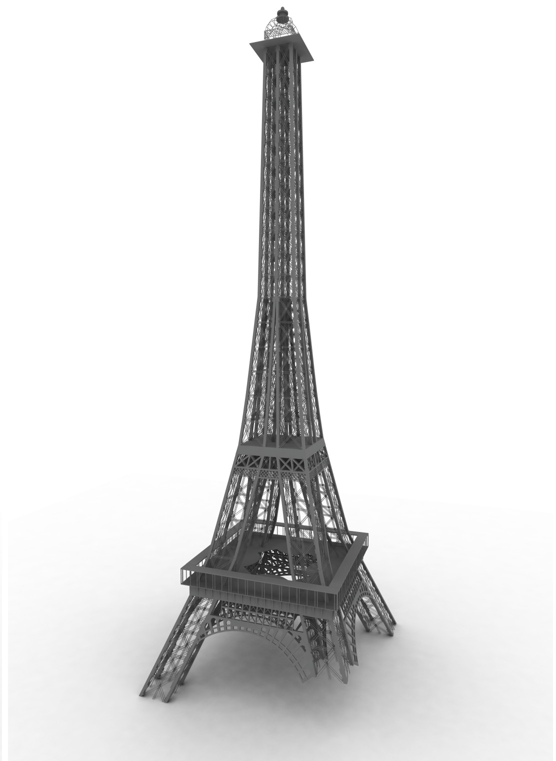
(a) A very simple variation, where only a part of the pillar is replaced



(b) A more complex variation, where the slopes are changed to give the Eiffel Tower a more ”sturdy” appearance.

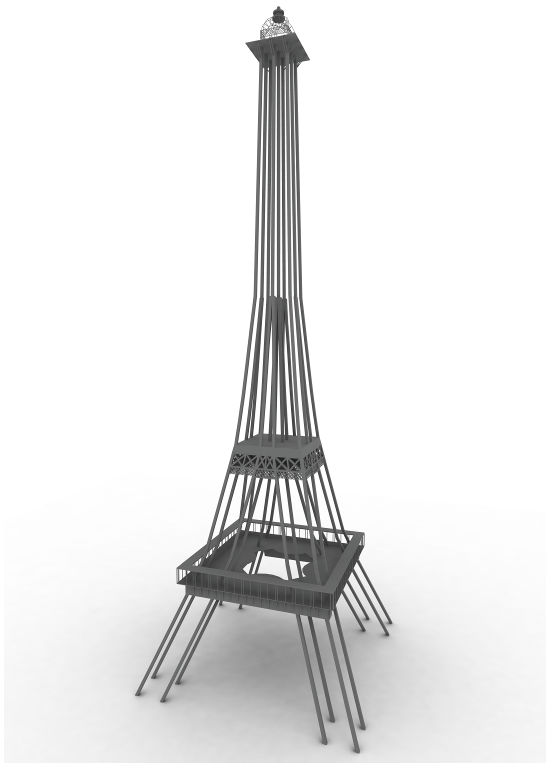


(c) Another variation that can be done is changing the different heights relative to each other. Note especially that the pillars meet at a much higher point than in the original model.

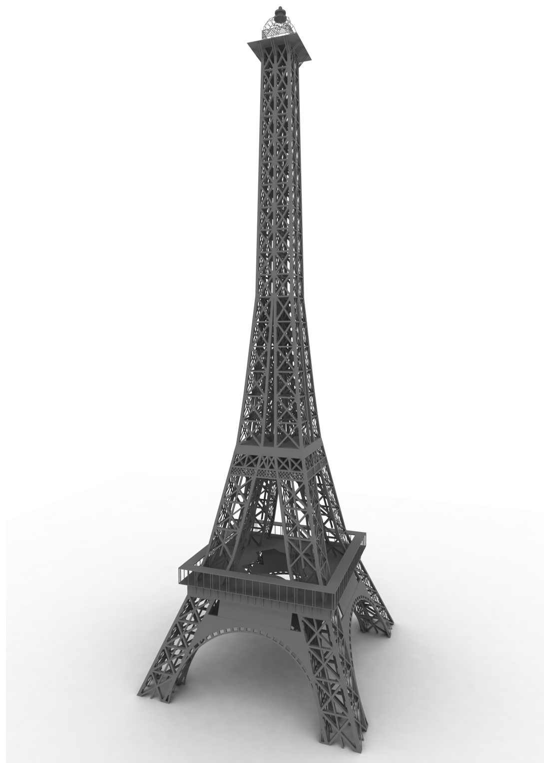


(d) What can also be interesting is to break the symmetry, for example by making one pillar shorter than the others. This could be used to place the Eiffel Tower on a non-planar surface, while still maintaining structural integrity.

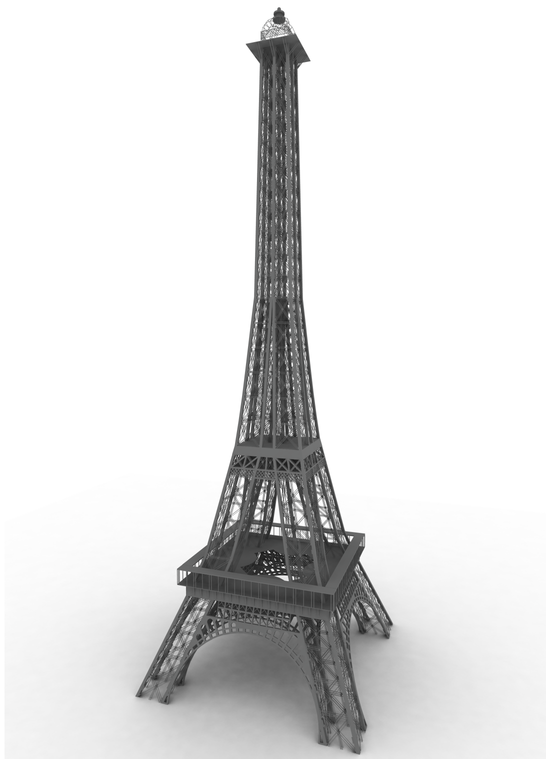
Figure 5.18: Variants of the Eiffel Tower



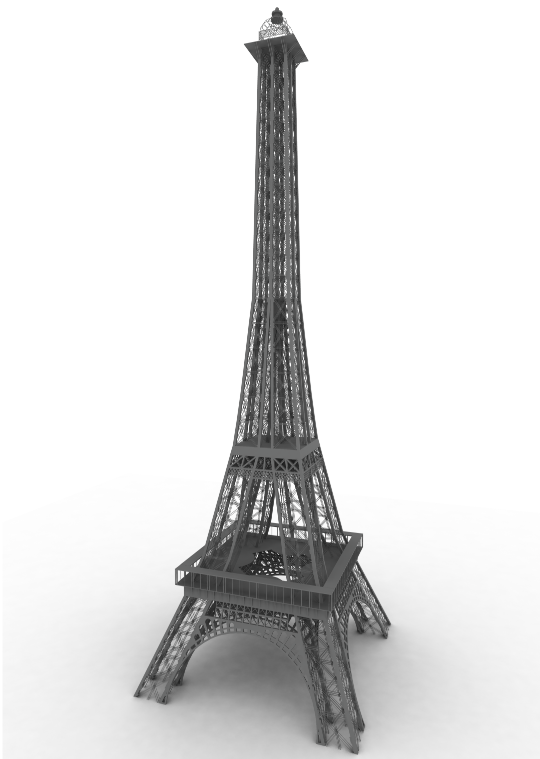
(a) Only the basic outline is present



(b) The basic outline is fit with a simplified lattice structure



(c) The lattice structure is refined



(d) Finally, rivets are added, which are only visible up close

Figure 5.19: The different levels of detail of the Eiffel Tower

Chapter 6

Conclusion

6.1 Summary

An extension for split grammars was presented, which can be used to easily model interconnections between model parts. It works by allowing each rule to have a local sub-rule that can connect multiple non-terminals of that sub-tree. By keeping the rules local, both implementation complexity and difficulties in usage are minimized. To allow these rules to perform useful operations on the selected scopes, several grammar functions in addition to simple splits have been defined.

Locally context-sensitive sub-rules were also added to an existing split grammar implementation based on the language GML. The implementation maintains a hierarchy of scopes, and provides functions to perform the lookups necessary to select multiple scopes for the left hand-side of a grammar rule. The additional grammar functions were also implemented, and can be used on the selected scopes, to create new ones and apply further grammar rules to them. Additionally, a simple, semantic level-of-detail concept was integrated with the grammar implementation.

To demonstrate the feasibility of this approach, several examples are provided, which include simple tile arrangements and connected planks. As a proof of the full power of the extension, the Eiffel Tower was modeled, down to the individual rivets. Several substructures were highlighted to show how and where the extension is actually needed.

6.2 Limitations

While the Eiffel Tower certainly looks impressive, there are several limitations, and certain things that still can not be modeled easily. Since the symbol lookups are performed by name rather than by location, some kinds of connections are difficult to model, for example connecting two sides of a gallery by the shortest possible bridge. In the current implementation, it would be necessary to explicitely state which parts of the gallery are to be linked.

Another issue are dependencies between parameters. In the current implementation it is very easy to

accidentally introduce such dependencies, and when one of the parameters is changed later, the model ”breaks”. For example, in the model of the Eiffel Tower, the various angles of the support beams actually depend very strongly on one another. Since the basis for the model was the book containing blueprints for all parts, these angles could be chosen correctly, but it is very hard to change them retroactively, and create different variations of the model in that regard.

Finally, usability is still a concern, too. Limitating context-sensitivity to local parts of the tree certainly helps, and allows the development of parts in a self-contained way, while guaranteeing that later integration will actually work. However, some connections, and usually the most prominent ones, are actually defined on the highest level, therefore requiring to be ”local” to the start-rule anyway. It is then tedious to always require everything to be constructed just to see that a big arch is not placed properly. A well thought out definitions of the different levels of detail reduces this issue somewhat, and the level-of-detail support was actually mainly added to aid development. As with all complicated design processes, the first model needs a lot more time to be completed than subsequent ones, since many ”tricks” and patterns have to be learned first. However, the addition of context-sensitivity feels natural, and once its potential has been grasped, allows for very elegant solutions for some problems.

What context-sensitivity can not provide, though, is to retroactively change the structure of sub-parts. For example, it is not possible to change split intervals based on what a structure is connected with.

Since the grammar is actually written as code in a programming language, most of these problems can be worked around by leaving the confines of the shape grammar formalism and simply write code. Of course, being able to do this is an advantage of this approach, but it is not always an option, especially in reverse modeling tasks. Therefore, care must be taken that the formalism is expressive enough for a wide range of problems.

6.3 Future work

To mitigate the limitations discussed above, several improvements can still be made on the basic principle. It is certainly possible, to also include geometric queries, rather than just allowing the selection of sub-scopes by name. In fact, some of the provided grammar functions, like the CSG-operations, already fulfill this rule to some extend. Therefore, other such grammar functions could be provided for specific requirements. The selection process could also be made more general by providing the ability to select scopes matching some parameters.

On the other hand, solving the problem with parameter dependencies is a much harder problem. Since parameters can be defined and used for anything, it is impossible for a computer to decide if something is a dependency or just a coincidence. Here, further research into how to annotate and restrict dependencies, and resolve unwanted ones would be required.

Improving usability is also not an easy endeavor. As already stated in [15], basically all existing im-

plementations suffer from the same issues in one way or another. The implementation presented here has the additional issue that it is not a ”pure” shape grammar, but basically a language extension and also restriction, so all the issues that come with programming are also present. The grammar representation is fairly regular, though, and it is therefore easy to imagine a specialized tool to be implemented, with which only code strictly conforming to the rules of the shape grammar can be produced, and subsequently also edited again.

Another possible extension to the presented method comes from the fact that the evaluation proceeds lazily. In every layer of the tree, the information about what operation was performed and which subscopes resulted from that operation, is stored. Any subsequent pass over this tree can then manipulate these parameters and even re-evaluate them, effectively allowing to change splits retroactively. Simple experiments, where a stairwell is moved depending on where a column was placed, have already been performed, but a remaining question is how this fits with the general framework of the shape grammar.

Bibliography

[1] Manish Agarwal and Jonathan Cagan. A blend of different tastes: The language of coffee makers. 1996.

[2] Christopher Alexander et al. *A city is not a tree*. utg., 1974.

[3] Autodesk. 3d studio max.

[4] Autodesk. Maya.

[5] Farès Belhadj and Pierre Audibert. Modeling landscapes with ridges and rivers: bottom up approach. In *Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 447–450. ACM, 2005.

[6] Scott Chase. Shape grammar implementations - the last 35 years, 2010.

[7] Hau H Chau, XIAOJUAN Chen, ALISON McKAY, and ALAN de PENNINGTON. Evaluation of a 3d shape grammar implementation. *Design computing and cognition*, 4:357–376, 2004.

[8] Bernard Chazelle and David P Dobkin. Optimal convex decompositions. *Computational Geometry*, 4(5):63–133, 1985.

[9] Noam Chomsky. *Syntactic structures*. de Gruyter Mouton, 2002.

[10] Rodrigo Coutinho Correia, José Pinto Duarte, and António Menezes Leitão. Malag: a discursive grammar interpreter for the online generation of mass customized housing. In *Proc. 4th Int. Conf. Design Computing and Cognition*, 2010.

[11] Ulrich Flemming. More than the sum of parts: the grammar of queen anne houses. *Environment and Planning B: Planning and Design*, 14(3):323–350, 1987.

[12] Institute for Computer Graphics and TU Graz Knowledge Visualization. Gml homepage, `http://www.generative-modeling.org/`.

[13] Institute for Computer Graphics and TU Graz Knowledge Visualization. Gml wiki, `http://hydra.cgv.tugraz.at/gmlwiki/index.php?title=Main_Page`.

[14] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, 1982.

[15] James Gips. Computer implementation of shape grammars. In *NSF/MIT Workshop on Shape Computation*, 1999.

[16] Bertrand Lemoine Gustave Eiffel. *The 300 Meter Tower*. Taschen Verlag, 2008.

[17] Sven Havemann and Dieter W Fellner. Generative mesh modeling.

[18] Bernhard Hohmann, Sven Havemann, Ulrich Krispel, and Dieter Fellner. A gml shape grammar for semantically enriched 3d building models. *Computers & Graphics*, 34(4):322–334, 2010.

[19] Bernhard Hohmann, Ulrich Krispel, Sven Havemann, and Dieter Fellner. Cityfit: High-quality urban reconstructions by fitting shape grammars to images and derived textured point clouds. In *Proceedings of the 3rd ISPRS Workshop*. Citeseer, 2009.

[20] J Mark Keil. Decomposing a polygon into simpler components. *SIAM Journal on Computing*, 14(4):799–817, 1985.

[21] Alex D Kelley, Michael C Malin, and Gregory M Nielson. *Terrain simulation using a model of stream erosion*, volume 22. ACM, 1988.

[22] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948. IEEE, 1995.

[23] Joan L Kirsch and Russell A Kirsch. The structure of paintings: formal grammar and design. *Environment and Planning B: Planning and Design*, 13(2):163–176, 1986.

[24] Terry W Knight. Shape grammars: six types. *Environment and Planning B*, 26:15–32, 1999.

[25] TW Knight. Color grammars: designing with lines and colors. *Environment and Planning B: Planning and Design*, 16(4):417–449, 1989.

[26] Lars Krecklau and Leif Kobbelt. Procedural modeling of interconnected structures. In *Computer Graphics Forum*, volume 30, pages 335–344. Wiley Online Library, 2011.

[27] Ramesh Krishnamurti. The arithmetic of shapes. 1980.

[28] Ramesh Krishnamurti. The construction of shapes. 1981.

[29] Ramesh Krishnamurti. Sgi: a shape grammar interpreter. *User Manual The Open University (Milton Keynes, England)*, 1982.

[30] IK Li and LM Kuen. A set-based shape grammar interpreter, with thoughts on emergence. In *First International Conference on Design Computing and Cognition Workshop*, 2004.

[31] Markus Mathias, Andelo Martinovic, Julien Weissenberg, and Luc Van Gool. Procedural 3d building reconstruction using shape grammars and detectors. In *3D Imaging, Modeling, Processing, Visualization and Transmission (3DIMPVT), 2011 International Conference on*, pages 304–311. IEEE, 2011.

[32] William J Mitchell. *The logic of architecture: Design, computation, and cognition*. MIT press, 1990.

[33] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. *Procedural modeling of buildings*, volume 25. ACM, 2006.

[34] F Kenton Musgrave, Craig E Kolb, and Robert S Mace. The synthesis and rendering of eroded fractal terrains. In *ACM SIGGRAPH Computer Graphics*, volume 23, pages 41–50. ACM, 1989.

[35] Yoav IH Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308. ACM, 2001.

[36] Przemyslaw Prusinkiewicz, Aristid Lindenmayer, and James Hanan. Development models of herbaceous plants for computer imagery purposes. In *ACM SIGGRAPH Computer Graphics*, volume 22, pages 141–150. ACM, 1988.

[37] Przemyslaw Prusinkiewicz, Aristid Lindenmayer, James S Hanan, F David Fracchia, Deborah R Fowler, Martin JM de Boer, and Lynn Mercer. *The algorithmic beauty of plants*, volume 2. Springer-Verlag New York, 1990.

[38] Ken Shoemake. Animating rotation with quaternion curves. *ACM SIGGRAPH computer graphics*, 19(3):245–254, 1985.

[39] Ondrej Št'ava, Bedrich Beneš, R Měch, Daniel G Aliaga, and Peter Krištof. Inverse procedural modeling by automatic generation of l-systems. In *Computer Graphics Forum*, volume 29, pages 665–674. Wiley Online Library, 2010.

[40] George Stiny. Two exercises in formal composition. *Environment and Planning B*, 3(2):187–210, 1976.

[41] George Stiny. Introduction to shape and shape grammars. *Environment and planning B*, 7(3):343–351, 1980.

[42] George Stiny. Weights. *Environment and Planning B*, 19:413–430, 1992.

[43] George Stiny and James Gips. Shape grammars and the generative specification of painting and sculpture. *Information processing*, 71(1460-1465), 1972.

[44] George Stiny, William J Mitchell, et al. The palladian grammar. *Environment and Planning B*, 5(1):5–18, 1978.

[45] Olivier Teboul, Iasonas Kokkinos, Loıc Simon, Panagiotis Koutsourakis, and Nikos Paragios. Shape grammar parsing via reinforcement learning. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 2273–2280. IEEE, 2011.

[46] .theprodukkt. .kkrieger.

[47] Benjamin Watson, Pascal Muller, Peter Wonka, Chris Sexton, Oleg Veryovka, and Andy Fuller. Procedural urban modeling in practice. *Computer Graphics and Applications, IEEE*, 28(3):18–26, 2008.

[48] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. *Instant architecture*, volume 22. ACM, 2003.