TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

MASTERARBEIT

Interactive Computer Generated Architecture

Ausgeführt am
Institut für Computergrafik und Algorithmen
der Technischen Universität Wien

unter der Anleitung von
Univ.Prof. Dipl.-Ing. Dr.techn. Werner Purgathofer
und
Univ.Ass. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
als verantwortlich mitwirkenden Universitätsassistenten

durch

Markus Lipp

Breitenwangerstrasse 6
A-6600 Reutte

Wien, am 16. September 2007

Abstract

The creation of visually convincing architectural models using traditional modeling methods is a labor intensive task. Procedural modeling techniques strive to reduce the manual work an artist has to perform when modeling architecture. In previous work, those techniques were successfully applied to the creation of architecture. However, previous methods have a limited usability, as they are based on text editing. This makes modeling unintuitive and diminishes the advantages of procedural modeling.

Therefore, methods to interactively create procedural architecture, using a graphical user interface, are explored in this thesis. As interactivity also requires real-time rendering performance, methods to accelerate the generation of architecture are investigated. Further, the thesis provides a detailed report on previous procedural architecture generation techniques.

Kurzfassung

Die Erstellung von visuell überzeugenden Architektur-Modellen ist, wenn man traditionelle Methoden verwendet, sehr arbeitsaufwändig. Prozedurale Modellierungstechniken haben das Ziel, manuelle Arbeitsschritte zu reduzieren. Frühere Lösungen zeigten bereits funktionierende Anwendungen von solchen Techniken bei der Architekturerstellung. Allerdings waren diese Lösungen in ihrer Anwendungsfreundlichkeit beschränkt, da sie auf Text-Bearbeitung beruhen. Das macht Modellierung unintuitiv und verringert daher die Vorteile, die man durch prozedurale Techniken gewinnt.

Deswegen werden in dieser Masterarbeit Methoden vorgestellt, die eine interaktive Erstellung von prozeduraler Architektur, mit Hilfe eines grafischen User Interfaces ermöglichen. Weil eine interaktive Erstellung auch Echtzeit-Geschwindigkeit benötigt, werden auch Methoden, um die Architektur-Generierung zu beschleunigen vorgestellt. Ein weiter Beitrag dieser Masterarbeit ist eine detaillierte Übersicht von bisherigen prozeduralen Methoden zur Architektur-Generierung.

Contents

1. Introduction **7**

1.1. Architecture Modeling 8
1.2. Problems with Traditional Modeling 8
1.3. What is Procedural Modeling? 12
1.4. Production Systems 13
1.5. Problems with CGA 14
1.6. Thesis Objectives 16
1.7. Related Work 17
1.8. Terminology 20
1.9. Structure of this thesis 21

2. Shape Grammars in Architecture **22**

2.1. Basic Shape Grammar 22
2.1.1. Definitions and Examples 23
2.1.2. Implications 25
2.2. Set Grammars 26
2.3. Computer Implementations 27
2.4. Examples 28
2.5. Summary 28

3. Shape Grammars in Computer Graphics **30**

3.1. L-systems 30
3.1.1. L-System extensions 33
3.1.2. L-systems for building generation 34
3.2. Split grammars 34
3.3. Computer Generated Architecture (CGA) 35

3.3.1. Formal CGA Definition . . . 36
3.3.2. CGA commands . . . 44
3.3.3. Mass modeling . . . 50
3.3.4. Relation to L-Systems . . . 53
3.3.5. Comparison to Other Shape-Based Approaches . . . 54
3.4. Other Related Work . . . 56
3.5. Summary . . . 58

4. Interactive Computer-Generated Architecture **59**
4.1. Interactive Editor . . . 59
4.1.1. Overview on Graphical Interaction . . . 60
4.1.2. Underlying concepts . . . 68
4.2. Language Enhancements . . . 72
4.2.1. Subtree Parameters . . . 72
4.2.2. Production Hierarchies . . . 74
4.2.3. Volumetric shapes . . . 76
4.3. Summary . . . 78

5. Implementation **79**
5.1. Software Architecture . . . 79
5.1.1. System overview . . . 79
5.1.2. Parsing and Semantic Analysis . . . 80
5.1.3. Production Application . . . 84
5.2. Performance Optimization . . . 84
5.2.1. Hardware-Accelerated Occlusion Queries . . . 85
5.2.2. Object Pools . . . 86
5.2.3. Flyweight Design Pattern . . . 87
5.2.4. OpenGL pseudo-Instancing . . . 89
5.2.5. Optimized Sorting . . . 89
5.3. Summary . . . 90

6. Evaluation **91**
6.1. Overall Performance . . . 91
6.2. Impact of Optimizations . . . 96
6.3. Summary . . . 99

7. Conclusion and Outlook **100**

7.1. Conclusion 100

7.2. Outlook 101

A. Formal language definition **104**

B. Bibliography **112**

Chapter 1.

Introduction

The creation of content for virtual worlds in computer games or movies is a very time consuming process. This process can require several man years worth of labor [MWH+06]. As the graphical power of computers increases strongly, the amount of content needed for state-of-the-art graphics in computer games increases too. This makes content creation even more expensive, and will continue to do so in the future.

Therefore, it is important to investigate methods to reduce the cost associated with content creation. The main idea is helping content-creation artists by providing methods to automate common modeling steps. Those methods are called *procedural modeling* techniques. This thesis focuses on techniques to simplify architecture creation. One recently proposed method is Computer Generated Architecture (CGA) by Müller et al. [MWH+06]. CGA is able to generate visually convincing building models, and is used as base for my thesis. Strengths and weaknesses of CGA will be investigated, and methods to alleviate some disadvantages are proposed.

The main disadvantage of CGA is limited usability, as it is based on text editing. This diminishes the cost advantages procedural methods can provide, as unintuitive textual definition of architecture is necessary. Therefore the main goal of this thesis is describing methods to graphically utilize some CGA functionality. Further, CGA is not targeted at real-time editing of buildings. However, graphical interaction methods need direct visual feedback to be usable. Therefore, methods to accelerate CGA in order to enable real-time editing and rendering are investigated.

In this chapter we will give an introduction to architecture modeling techniques. At first, the traditional modeling method is described. Then the problems with this approach are pointed out. In Section 1.3 procedural modeling techniques, solving some of those problems, are explained. Afterwards, production systems, a specific procedural technique, are investigated in more detail. Then we point out some problems of production systems in Section 1.5. This leads us to the objectives of this thesis, explained in Section 1.6. After that, the related work of this thesis is discussed. We will have a look at the terminology used in this thesis in Section 1.8. Finally, the structure of this thesis is pointed out.

1.1. Architecture Modeling

Traditionally, standard 3D modeling packages, like Autodesk 3D Studio Max or Maya, are used to create visually convincing computer models of architecture. Those packages offer many different tools to manipulate geometry, and therefore most building styles can be created with such programs. The main disadvantage is the high amount of manual work required to get to a convincing result [MWH+06]. This leads to high costs when many buildings, or whole cities, need to be modeled.

This is where Computer Generated Architecture (CGA) [MWH+06], a procedural modeling technique, comes into play. CGA strives to reduce the cost associated with the modeling of buildings. To understand how this is achieved, let us point out the problems with the traditional approach at first.

1.2. Problems with Traditional Modeling

We will now point out different problematic areas when buildings are created using standard tools. The goal is to analyze why the traditional modeling is a costly process. A short description how CGA can help in different areas is provided. Note that we only roughly point out what CGA can do, a detailed description is given in the following chapters.

Figure 1.1.: When a ground plan vertex is dragged with the mouse, the windows are automatically distributed.

Manual Distribution: When different building components, for example walls and windows, are designed, they need to be placed many times on façades. While some tools help in the placement, those tools need to be manually applied everytime a façade's size changes. CGA has a very powerful method to automate this step: Simple *production rules* define how the placement should be performed. Now, according to the production rules, all components are distributed automatically on the façades. Figure 1.1 illustrates the solution using CGA.

No High-Level View: In a traditional modeling system, components of a building are modeled individually and then distributed. There is no view on the building as a whole, for example to set high-level parameters of a building. In CGA it is possible to treat the building as an

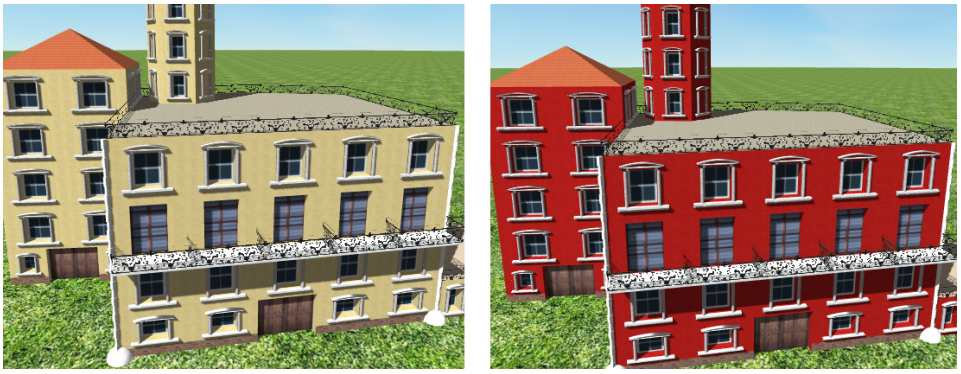

Figure 1.2.: Because the house is an entity of its own, we can change the texture of all wall tiles with just one drag-and-drop operation.

entity: You can set parameters or textures for the whole building, without manually modifying the individual components. Figure 1.2 shows one application of this approach.

Iterative Process: The previous two points implicate problems when trying to use an iterative design process: For example, when the user changes the ground plan of a building, all building components need to be manually distributed again. Using CGA, a fully iterative process is possible: Changing of the ground plan automatically rearranges all components. An example iterative modification is shown in Figure 1.3.

Reusability: Parametrization and flexible combination of building components are the keys to reusability. In traditional packages, it is often not possible to assign parameters to an building component. Further, combinations of components have to be performed manually. On the other hand, CGA offers full parametrization possibilities of CGA production rules. By interleaving multiple CGA production rules, the components can be combined too. A parametrized component is shown in Figure 1.4.

Now that we know where CGA can help, we will describe how this is actually possible. Therefore, procedural modeling has to be explained.

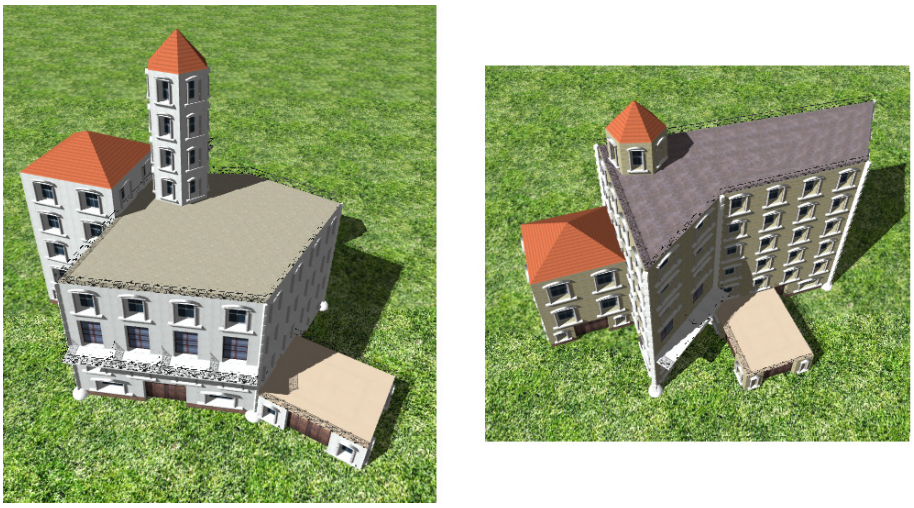

Figure 1.3.: The iterative process allows rapid adjustments of buildings. On the left side, the original is shown. On the right side, textures, some parameters and the ground plan were modified. All modifications were conducted in about one minute.

Figure 1.4.: On the left side, the original window is shown. The column is combined with the window, and can be flexibly interchanged with other columns. The window has some parameters, like ornament height. On the right side, another column type is used, and ornament parameters are changed, using sliders and drag&drop operations. Those modifications took about one minute.

1.3. What is Procedural Modeling?

Defining procedural modeling is a difficult task. As Havemann points out [Hav05]: "The attribute 'procedural', however, is a bit vague, since this term is used in too many different contexts." During research for this thesis, we found this statement to be appropriate. To lessen the confusion surrounding this term, we will provide an overview on procedural modeling, as used in the context of building generation.

Procedural Modeling, Literally One approach emerges, when the term "procedural" is taken literally as in "procedural programming": The generation of geometry is done in an imperative programming language. Geometry is created by calling procedures on data, loops and conditionals are allowed. One example of such an approach is the Generative Modeling Language (GML) [Hav05]. In GML, a stack-based programming language, shown to be turing complete, generates geometry. This way, theoretically all possible kinds of geometry can be generated [Hav05].

Geometry generation is performed on a low level, starting with triangles. There are no high-level constructs to capture and distribute architectural concepts. Therefore it is questionable whether this approach is suitable for the modeling of buildings: Instead of designing high-level aspects of a building, the artist would need to create a building starting from triangles.

An interesting variation of this approach is to employ a visual language for the imperative language constructs: Instead of writing code, visual symbols are connected. This method is described by Ganster and Klein [GK07].

Procedural Modeling as a Stack of Operations Another take on "'procedural"' is to define it as a stack of multiple, subsequent geometric operations: Instead of saving the result of a geometric operation like extrusion, only the functions with parameters that need to be called on the input data, are saved. This stack of operations can then be applied to different input data. Note that this is a special case of "Procedural Modeling, Literally": Imperative constructs like loops and conditionals are not permitted. This method is shown to be applicable on components of buildings, like columns and windows [BBJ+01]. However, the problem how

to distribute those building components on façades to create whole buildings is not solved by Birch et al. in their publication [BBJ+01].

Production Systems The third possibility of procedural modeling are production systems. We will provide a detailed look on them in the next section.

1.4. Production Systems

CGA is a production system. There are also other production systems, for example L-systems. A very good introduction and comparison of production systems is provided by Gips and Stiny [GS80]. In this section we will provide an abstract definition of production systems. In the following chapters, specific production systems (Shape grammars, L-Systems, Split Grammars and CGA) are introduced.

At first, we will summarize the key findings of Gips and Stiny, combined with the application of those findings to shape grammars by Wonka et al [WWSR03]. Then an example is provided.

Definitions Production systems are defined for specific *Objects*. For example, L-systems are based on strings, and CGA is based on shapes. The set of objects a specific system operates on is called the *vocabulary* U. The vocabulary consists of terminal symbols $\Sigma \subseteq U$ and nonterminal symbols $V \subseteq U$.

A production system contains a list of *productions* R in the form $v \rightarrow u$, $R \subseteq V \times U^+$. Objects are produced by starting with an initial object I, and repeatedly applying productions. This is called the *production process*. The initial object is an arbitrary combination of objects from the vocabulary, and is called *axiom*.

Productions can be *applied* when some transformation $f(v)$ of v occurs in the current object w, formally $f(v) \leq w$. The result of an application is the replacement of $f(v)$ with $f(u)$ in the current object, formally $[w - f(v)] + f(u)$. Possible transformations f are dependent on the object type, for example, when shapes are used, f can be a scaling transformation.

The tuple (V, Σ, R, I) is called a *grammar*. Please note that in Instant Architecture [WWSR03] V is named N and Σ is named T. The terminology V and Σ was introduced in the more recent publication on CGA [MWH$^+$06].

Example In order to make the previous paragraph more clear, we will provide an example shape grammar.

Figure 1.5 visualizes a grammar consisting of a vocabulary (V, Σ), productions R named A and B, and an axiom. The axiom is a transformation $f(v)$, as the size is different from v. Please note how this transformation is also applied to the result u of the production A in the first application step.

The production process starts with the axiom, then production A is applied. Production B is applied two times. This example is a simplification to illustrate the concepts, a more rigid example is found in Chapter 2.

1.5. Problems with CGA

Until now, we have only talked about the benefits of CGA and production systems. Of course there are also disadvantages. The main goal of this thesis is to identify disadvantages and try to alleviate them.

CGA, as proposed by Müller et al. [MWH$^+$06] has three main disadvantages:

- Usability: The production rules that define distributions of shapes are text based. Every modification and every parameter needs a change in the text file. This complicates building modifications and is not intuitive for a modeler.
- Artist control: As a consequence of the first problem, an artist does not have direct local control over the generated output. It would be advantageous if the artist could override some decisions the production system made, using a graphical editor.

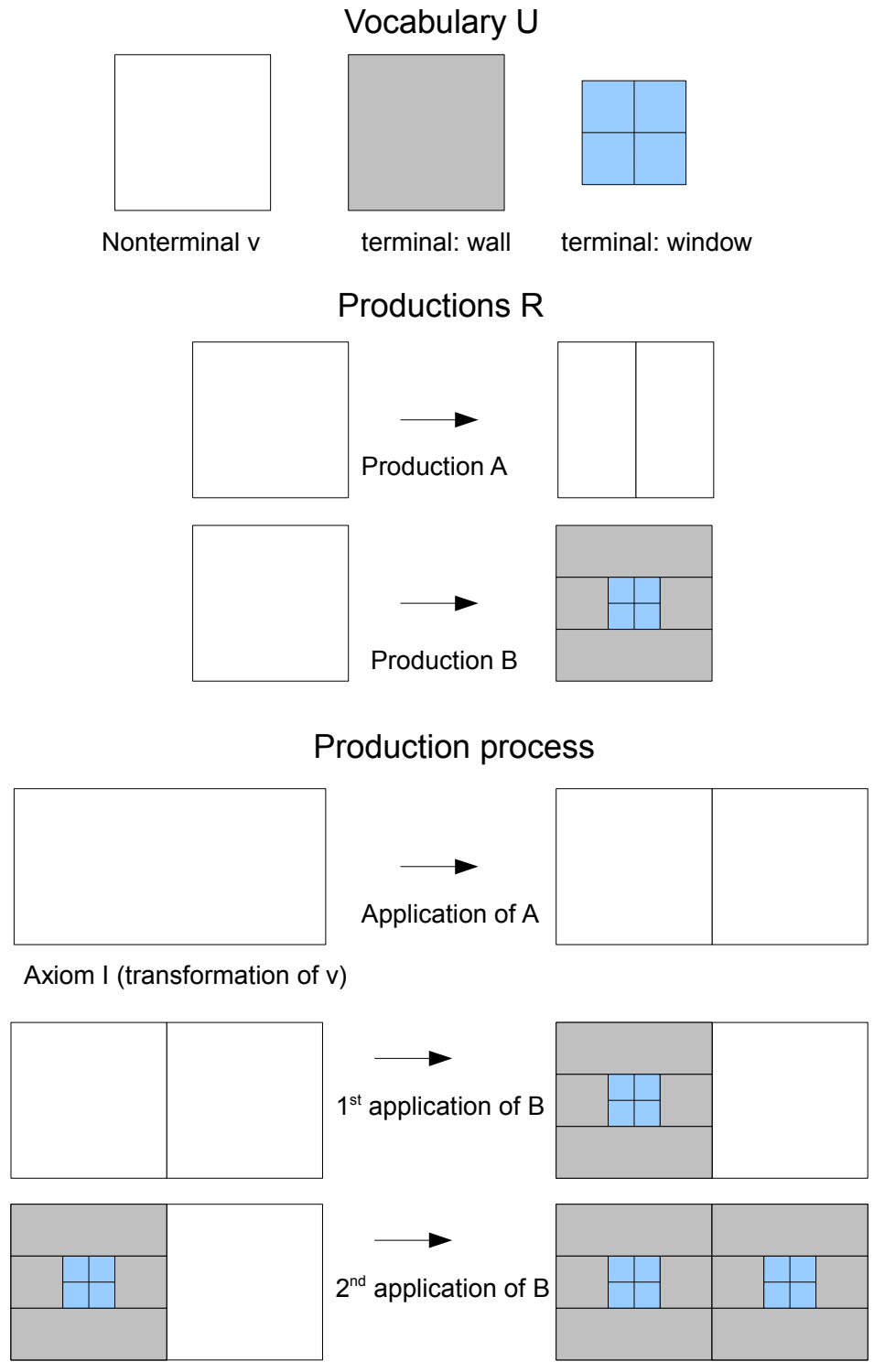

Figure 1.5.: Visualization of a grammar and the production process

- Performance: When a text based editing system is used, performance of building generation is not that critical. Therefore, Müller et al. do not provide details on possible performance improvements in their publication [MWH+06]. However, when we want direct graphical modification of buildings, the generation performance must be in real time, to enable direct visual feedback. We consider real time to be at least 25 building generations per second.

The first two can be alleviated by providing an interactive graphical editor, the last one requires detailed explanations on performance improvement techniques.

1.6. Thesis Objectives

As the problems of previous approaches were described, we can now outline the three main objectives:

1. Create a complete CGA implementation from scratch, as we do not have a code base to work on.
2. Increase the usability of CGA
    - Allow interactive editing
    - Provide artist control
3. Show techniques to increase the performance of CGA to allow real-time manipulation of parameters

Point one and three are subject of Chapter 5. Point two is evaluated in Chapter 4. In the Chapter 6 we will evaluate if our objectives were reached.

1.7. Related Work

The main papers this thesis is based upon, Procedural Modeling of Buildings [MWH+06], Instant Architecture [WWSR03] and L-systems [PL96], are described in detail in Chapter 3. The basics of shape grammar in architecture are described in Chapter 2. Other relevant work will be described in this section.

A few research topics are related to CGA. Some of them are important if CGA should be used in a larger scope, for example, to generate whole cities. We will now point out these areas and provide a list of the most important publications on those topics. Figure 1.6 shows a graphical representation of adjacent areas. Arrows represent dependencies between domains. Note that those adjacent topics will *not* be covered in this thesis, only building generation is investigated. In this thesis, the building lots are assumed to be already existent.

Landscape Manual creation of landscapes is possible by drawing a height field representation. There are many different methods to automate this step. One example is the midpoint displacement algorithm [FFC82]. An example landscape generated using this method is shown in Figure 1.7. Another possibility is to use elevation data from existing landscapes. As those topics only have an indirect influence on building generation, we will not go into further detail.

Zoning, Street Generation, Lot Generation Those three areas are often solved using combined approaches. Let us describe the areas at first, and then provide an overview on proposed approaches.

Zoning describes the distribution of building types (commercial, residential) and semantic tags (wealthiness, population density) across a landscape [PM01a], to define high-level properties of a city. Lot generation, also called allotment, creates the actual building footprints to be used for building generation. Street generation creates networks of roads in a defined area.

Street generation is tied to allotment. Some algorithms generate streets and create lots along those streets [FWB+01] [PM01b]. Others are only concerned with street layout generation

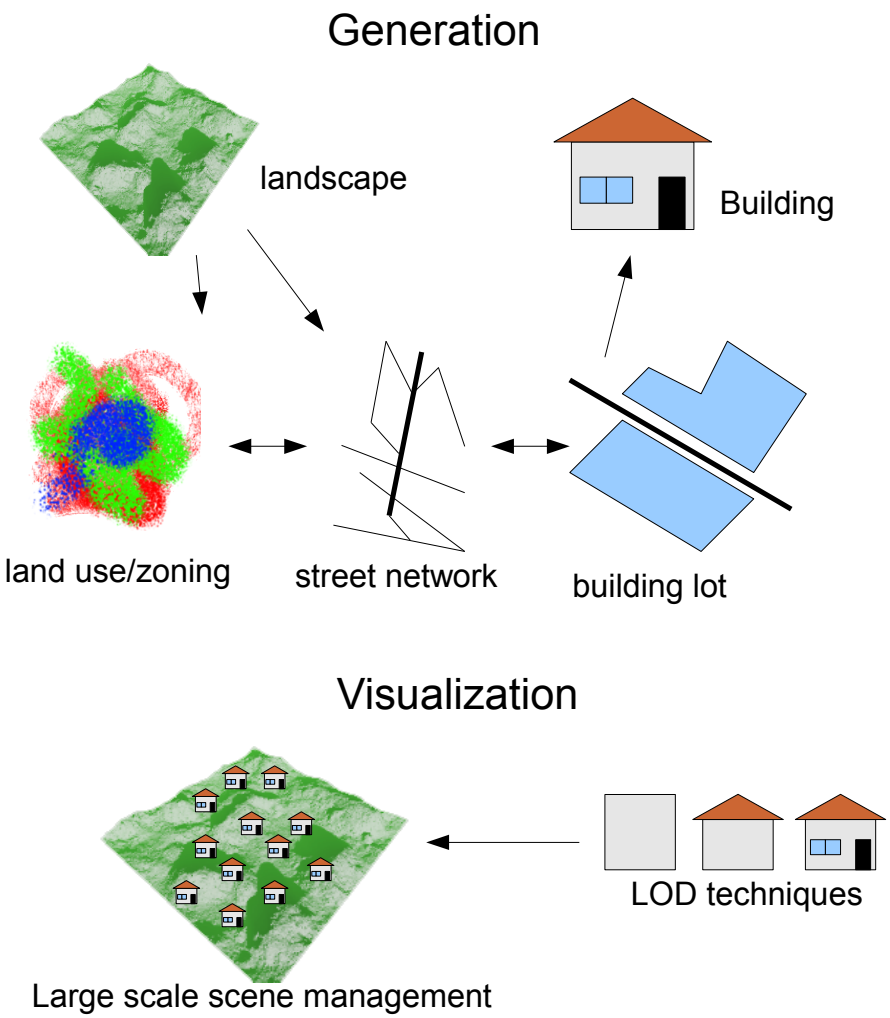

Figure 1.6.: Building generation in context of other operations is shown on the top. Aspects important for visualization are shown below.

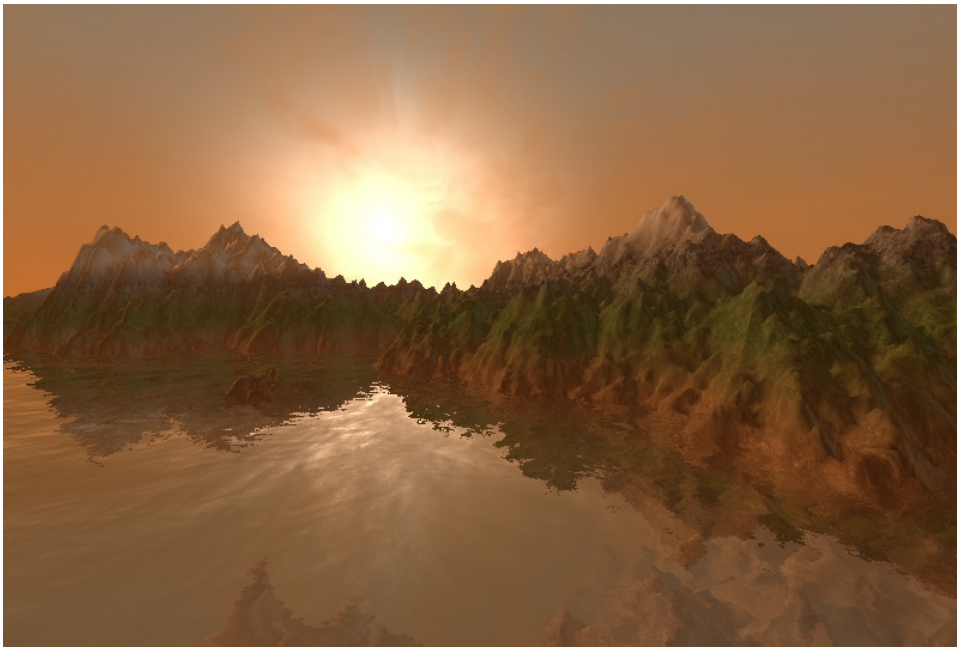

Figure 1.7.: Landscape generated using midpoint displacement. The screenshot is from an application written by me for another lecture.

[GMB06]. Also, dedicated allotment algorithms, utilizing no street generation, were proposed [dSM06] [CM05] [LD03].

Zoning can be performed manually by drawing image maps representing the zones [PM01b], or automatically using simulation algorithms [LWWF03].

Large Scale Scene Management The handling of many objects, along with the assembly of such scenes, is important when whole cities need to be generated. This thesis focuses on the generation of single buildings, and thus those problems do not arise. Scene assembly is presented by Flack et al. [FWB+01], the management problem is evaluated by Silveira and Musse [dSM06].

Levels of Detail To get acceptable performance in real-time applications when using complete city models, LOD methods need to be used for buildings: Distant buildings are drawn with a coarser detail level. Again, as this thesis focuses on single buildings, LOD methods

are not necessary. However, creating LOD techniques for CGA is an interesting research problem for future work.

Different LOD techniques applied to whole cities were proposed [DB05] [dSM06].

1.8. Terminology

The publications describing production systems use a slightly different choice of words when describing similar things. To avoid confusions, we define the terminology used in this thesis here. Our terminology is based on the one used by Müller et al. [MWH+06].

Production Rule A production rule describes a replacement of a generic object with other generic objects. For example, those generic objects can be shapes or strings.

Different production systems have slightly different terminology: In L-systems, production rules are called rewriting rules, in shape grammars they are just called rules. They are called production rules in CGA. In order to avoid these ambiguities, we always call them production rule, regardless of the context. As abbreviation of "production rule" we use "production".

When written out, productions have a predecessor object on the left hand side and successor objects on the right hand side. During application, the predecessor is replaced with the successor.

Production Application When we have a production rule and multiple objects, the replacement of every occurrence of the predecessor object with the successor object is called "production application".

Command This is the generic term for all functionalities that modify shapes (or generic objects) in some way. They are executed during production application. Examples are the split command or turtle commands, as described in later chapters. Please note that this term was not used by Müller et al. [MWH+06], we introduce this term to clarify the distinction between production rules and other functionalities.

Shape Shape is a specific object type. For example, in architecture shape is defined as an arbitrary arrangement of lines. Please note that production rules are not shapes themselves, they just have one shape as successor and one or multiple shapes as predecessor.

Production Application Process This is synonymous to "derivation process". It describes production applications starting from an axiom until no production rules are left, or an iteration threshold is exceeded.

Object During formal definitions, this term is used to point out generic properties. For example, in the grammar definition object refers to a generic type. However, when we speak of objects in the context of computer implementations, we refer to instances of classes, as used in object-oriented programming.

1.9. Structure of this thesis

We will now provide an overview of this thesis:

- Chapter 2 looks at the roots of the shape grammar formalism in architecture. Definitions as used in architecture are provided.
- Chapter 3 focuses on recent approaches on mapping shape grammars to computer graphics. This includes Instant Architecture and CGA.
- Chapter 4 proposes novel ideas to improve previous shape grammar implementations.
- Chapter 5 provides a detailed description of implementation issues and details.
- Chapter 6 evaluates the performance and the usability of the implementation.
- Chapter 7 summarizes the findings of this thesis.

Chapter 2.

Shape Grammars in Architecture

The shape grammar formalism was pioneered in architecture by Stiny and Gips [SG72]. In order to apply those concepts to computer graphics, it is necessary to understand the fundamental properties of this formalism. Therefore, this chapter will explain the shape grammar formalism.

One important thing to consider is the different goal of shape grammars in architecture versus computer graphics. On the one hand, in architecture, blue prints for buildings should be generated. Those blue prints need to have an accurate scaling. It is not the main goal to visualize the generated buildings. On the other hand, in computer graphics, a visually convincing building should be generated. The scalings are not that important, because accurate blue prints are not the main goal.

This chapter is structured as follows: At first, the basic formalism is outlined. Then set grammars are described. Afterward, computer implementations targeted at architects are examined. Finally, some examples of shape grammars applied in architecture are shown.

2.1. Basic Shape Grammar

Shape grammars were first defined by Stiny and Gips [SG72], later the formalism was refined by Stiny [Sti80]. This section will focus on those basic definitions. At first, definitions constituting the shape grammar formalism are summarized. Then, implications of those definitions are examined.

2.1.1. Definitions and Examples

We will simplify the definitions provided by Stiny [Sti80] and leave out parts that, in my opinion, do not directly lead to better understanding. For a detailed description please refer to the mentioned paper. Note that the formalism is conceptually the same as the one described in Section 1.4, however the nomenclature is slightly different.

Shapes are defined on a low level, as lines are the only allowed primitives:

Definition 1 *A shape is a limited arrangement of straight lines [Sti80].*

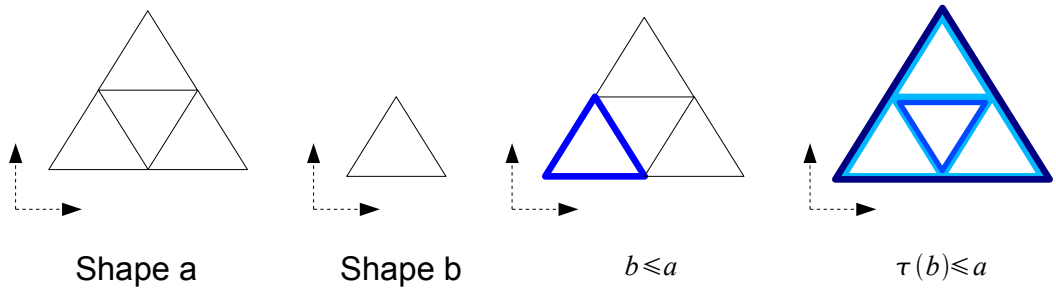

Figure 2.1.: Examples for shapes and subshape relations. Dashed arrows represent coordinate frames.

An example shape is depicted in Figure 2.1 on the left side. The notation of *subshapes* is required to specify which part of a shape is replaced in a production:

Definition 2 *A shape s_1 is a* subshape *of shape s_2 (denoted by $s_1 \leq s_2$) if and only if each line of s_1 is in s_2 (simplified version of [Sti80]).*

Intuitively, this corresponds to a pattern matching problem: One shape must be visually matched in another shape. The example subshape relation $b \leq a$ is seen in Figure 2.1. Please note that only one match occurs.

In order to specify when a production takes place, transformations of a shape must be defined:

Definition 3 *A transformation τ of a a shape s is the shape denoted by $\tau(s)$. These transformations are translation, rotation, reflection, scale or finite compositions of them [Sti80].*

When transformations are allowed on b, more subshape matches can be found, as seen in Figure 2.1 on the right side. The matches are highlighted in different shades of blue.

Further, boolean operations union, intersection and difference are allowed on shapes. Given a finite set of shapes as a vocabulary, we can form other shapes using transformations and boolean operations on this vocabulary. The resulting set is called S^+. When the empty shape s_0 is included, it is called S^*.

To distinguish shapes, we can add *labeled points* to shapes. A labeled point $p : A$ consists of a position p and a *symbol* A. To get a subshape match, the labels must match too. An example is seen in Figure 2.2. Both shape a and shape b have a labeled point C. There is no match for the subshape relation, and only one match for the subshape relation with transformations. As we can see, labels can decrease the number of possible matches.

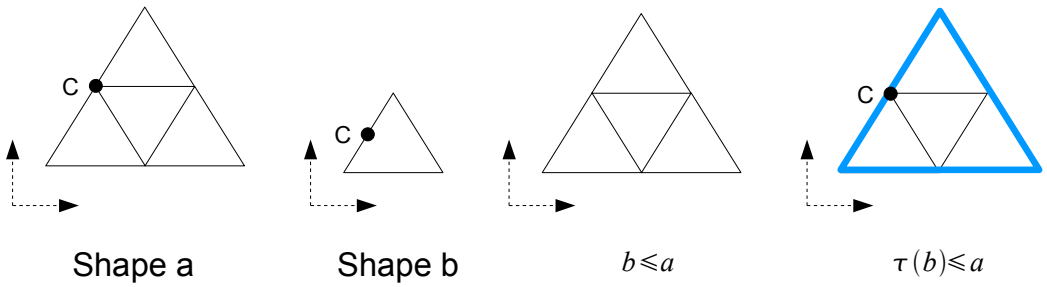

Figure 2.2.: Examples for labeled shapes and subshape relations.

Now, we have all notations to define a production rule:

Definition 4 *A production rule has the form $\alpha \rightarrow \beta$, where α is a labeled shape in S^+ and β is a labeled shape in S^*. A production rule* applies *to a shape γ when there is a transformation τ such that $\tau(\alpha)$ is a subshape of γ, that is, $\tau(\alpha) \leq \gamma$. Application of a production rule means replacing the occurrence of α with β, formally $(\gamma - \tau(\alpha)) + \tau(\beta)$ [Sti80].*

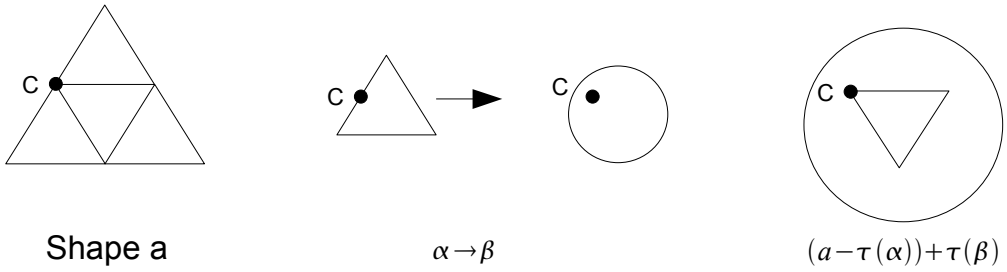

Figure 2.3.: Production rule with application.

Figure 2.3 displays an example production rule and an application of this production.

Using the previous definitions, we are now finally able to define shape grammars:

Definition 5 *A shape grammar has four components: (1) S is a finite set of shapes; (2) L is a finite set of symbols; (3) R is a finite set of production rules; (4) I is a labeled shape in $(S,L)^+$ called initial shape [Sti80].*

Applying a finite series of production rules to the initial shape yields in a new shape γ. The set of all shapes that can be generated this way is called the language of the grammar.

2.1.2. Implications

Shape definition is based on lines. Therefore, in order to find subshapes, line configurations need to be matched using arbitrary transformations. As an unlimited amount of possible transformations exist, this is a difficult task, and was shown to be undecidable [PF98]. This is referred as the *subshape problem*. It makes a computer implementation difficult.

Further, during production application, new shapes not included in S can be generated. Those shapes are called *emergent* shapes [Gip99]. Emergence further complicates the subshape problem.

Both emergence and the subshape problem imply the following: The creation of shapes and production rules, that create a specific intended result (for example a kitchen blue print), is difficult and requires a great deal of intuition by the designer [Cha89].

2.2. Set Grammars

In order to simplify the subshape problem, set grammars were introduced by Stiny [Sti82]. In this approach, shapes are *not* defined on the line level. Instead, shapes consist of *sets*. A set is an immutable group of lines and labels. Sets can be interpreted as symbolic objects.

Now, instead of looking at the lines to find subshapes, only a matching symbolic object needs to be found. This symbolic level makes computer handling of grammars easier. As the subshape matching is performed on a symbolic level, it is not possible for new sets to emerge. All sets occurring in a shapes derived through production application must occur in the vocabulary S [Sti82].

Set grammars and shape grammars define the same language of designs [Sti82]. However, in set grammars, we have to parse the desired design into different sets, and place them in the vocabulary. This is not necessary for shape grammars, as we directly operate on lines. We will adapt the previous example shown in Figure 2.3 to point out this difference.

Figure 2.4 shows how we parse the shape A into three sets, highlighted in different colors. These sets are immutable. Our resulting shape only consist of those sets. Of course we could also parse our shape into different sets, for example into three small triangles. This would generate a different result.

To sum this up, we can say that the big advantage of set grammars is the simplified subshape search. As a drawback, we have to decompose our design into different sets. However, this decomposition also has one advantage: We can exactly specify which sets can occur in our results. Unexpected sets can not occur.

Please note that all shape grammar based building generation algorithms, described in Chapter 3, are based on this simplified version of shape grammars, as this approach is easier to handle for computers.

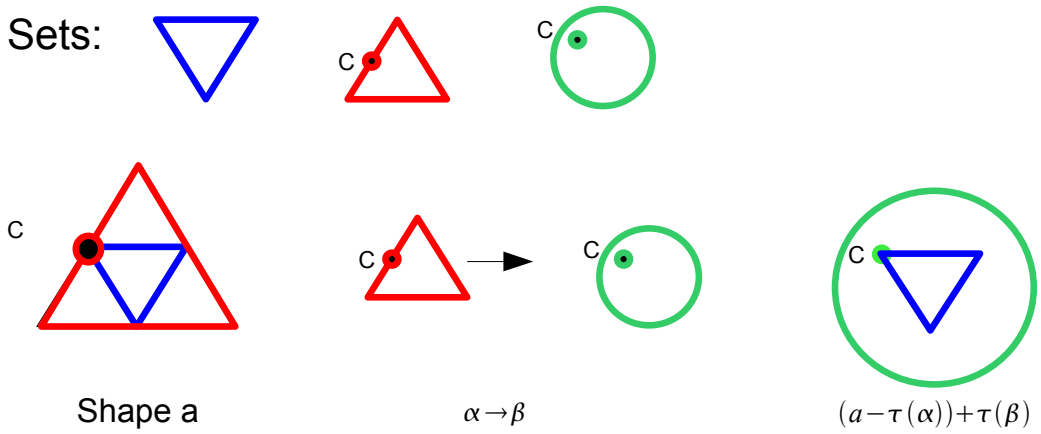

Figure 2.4.: Production rule application using set grammars.

2.3. Computer Implementations

Implementations targeted at architects have the purpose to simplify or automate some tasks associated with shape grammars. It is not possible to create complete visualizations of buildings with the proposed methods. A very good introduction is found in [Gip99], we will now summarize the key findings.

There are four areas of computer implementations:

- Given a production-rule set, a program can help in applying the productions. Either the user manually selects a production to apply, or the program automatically searches for matching subshapes. Those programs are called *interpreters*.
- A *parsing* program creates a sequence of productions to get from a specified shape set to a specific result
- An *inference* program automatically creates a shape grammar from a given set of results.
- A shape-based *Computer Aided Design* program combines the previous three areas. Such a program would allow designing production rules and helping in applications to generate blue prints.

Specific computer implementations were proposed by Chase [Cha89], Piazzalunga and Fitzhorn [PF98]. Please note that, to my knowledge, no method published in the field of architecture is actually able to generate visually convincing building models. Therefore we will not explain further details of those approaches, as they are not directly applicable to my thesis.

2.4. Examples

A complex shape grammar, consisting of 164 production rules, was introduced by Duarte [Dua05]. This grammar is able to generate mass customized housing.

Another example is the generation of mughul gardens by Stiny and Mitchell [SM80]. This grammar is able to generate complex gardens with watering systems. A result from our framework, using a simplified version of the mughul grammar, can be seen in Figure 2.5.

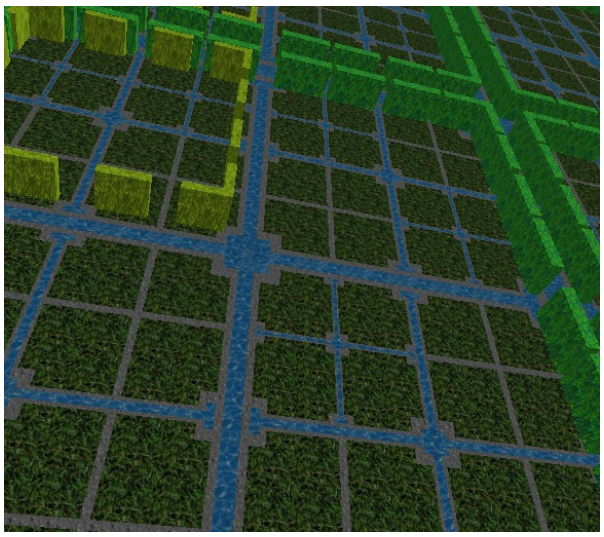

Figure 2.5.: Simplified mughul gardens

2.5. Summary

An explanation of the shape grammar formalism was provided, then the simplified set grammars were introduced. Afterwards computer implementations were described.

Two key points are important in this section:

- Shape grammars are based on arbitrary configurations of lines, and therefore hard to handle for computers. In order to alleviate this, set grammars can be used.
- Previous computer implementations aimed at architects are not able to generate visually convincing buildings.

This implies that the original shape grammar approach needs to be extended to be applicable for visually convincing renderings. The next chapter shows how this can be done.

Chapter 3.

Shape Grammars in Computer Graphics

In recent publications, the shape grammar formalism, originally targeted at architects, was applied to computer graphics in order to create visually convincing buildings. In this chapter we will summarize those publications.

At first, L-Systems are described, as they are the first grammar-based geometry generation approach introduced in computer graphics. Then, split grammars as introduced in Instant Architecture [WWSR03] are described briefly. Afterwards, a detailed description of CGA [MWH$^+$06] is provided. Finally, other related work is presented.

3.1. L-systems

L-systems are parallel string rewriting systems [PL96], mainly targeted at creating plants. Figure 3.1 shows an example tree generated using L-systems.

Given a start string ω and production rules (called rewriting rules in [PL96]) , every character in ω is replaced with the string of a matching production rule. Consider the following example, taken from [PL96]:

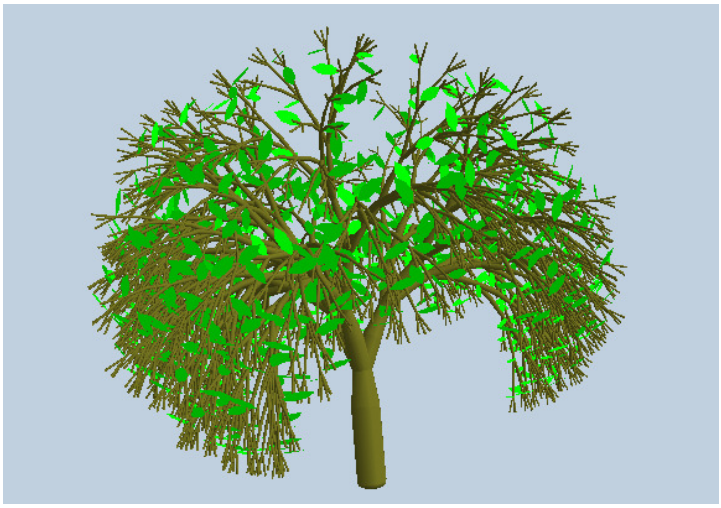

Figure 3.1.: Tree generated using L-system. The screen shot is from an application written by me for another lecture.

$$
\begin{aligned}
\omega : &\quad a_r \\
p_1 : &\quad a_r \rightarrow a_l b_r \\
p_2 : &\quad a_l \rightarrow b_l a_r \\
p_3 : &\quad b_r \rightarrow a_r \\
p_4 : &\quad b_l \rightarrow a_l
\end{aligned}
$$

Here, p_1 to p_4 are *production rules*. The left hand side of a production rule is called *predecessor*, the right hand side is the *successor* [PL96]. A character that matches the predecessor is replaced with the successor during one derivation iteration. The amount of iterations is user defined. For example, three iterations on the given system result in:

$$
\begin{aligned}
&a_r \\
&a_r b_l \\
&b_l a_r a_r
\end{aligned}
$$

An important property of L-systems is that geometry is not directly created during derivation. Instead, the resulting string has to be interpreted afterwards to create geometry. Therefore, specific characters represent the following commands, called *turtle commands* [PL96]:

- F: Move forward and draw a line
- f: Move forward without line drawing
- +: Rotate left
- -: Rotate right

Those are the most important commands. Other commands, extending L-Systems into 3D, are also available. When the derivation is finished, the string is searched sequentially for such commands, and the associated geometric action is performed.

The following L-system utilizes the described turtle commands, and generates a FASS curve [PL96]:

$$
\begin{aligned}
\omega &: \quad \mathbf{L} \\
p_1 &: \quad \mathbf{L} \rightarrow \mathbf{L}F\mathbf{R}F\mathbf{L} - F - \mathbf{R}F\mathbf{L}F\mathbf{R} + F + \mathbf{L}F\mathbf{R}F\mathbf{L} \\
p_2 &: \quad \mathbf{R} \rightarrow \mathbf{R}F\mathbf{L}F\mathbf{R} + F + \mathbf{L}F\mathbf{R}F\mathbf{L} - F - \mathbf{R}F\mathbf{L}F\mathbf{R}
\end{aligned}
$$

Please note that `L` and `R` are *not* turtle commands, they are predecessors of production rules. To make this distinction more evident, predecessors and their occurrences are typed in bold text. The result after three iterations is shown in Figure 3.2. When comparing the L-system with the generated output, a disadvantage of L-systems becomes obvious: It is not trivial for a human to predict the output of an L-system, and therefore hard to write productions creating an intended result.

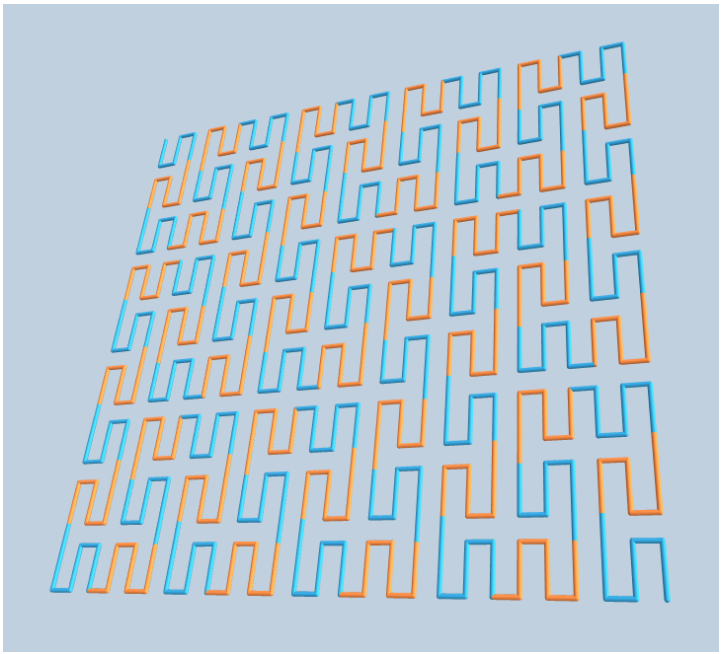

Figure 3.2.: FASS curve generated using L-system. The screen shot is from an application written by me for another lecture.

3.1.1. L-System extensions

Numerous extensions to the basic L-system formalism were proposed. We will summarize the most important ones.

Synthetic Topiary *Query modules* are introduced by Prusinkiewicz [PMM94]: They allow obtaining the current position. Those positions can then be fed into user defined functions. Such a function can for example calculate if the branch on the current position should be pruned. This way, plants can be pruned to specific shapes, creating a synthetic topiary.

Positional Information The usage of query modules is further extended by Prusinkiewicz [PMKL01]: Curves cane be defined externally using a graphical editor. Those curves can then be queried in the L-system, given a curve identifier and a relative position. Using this information, it is possible to directly specify silhouettes of plants. Modifications of the curves directly change the plant appearance, creating direct control for the artist.

3.1.2. L-systems for building generation

Split grammars and CGA share some concepts with L-Systems, as described in Section 3.3.4. However, one building generation method was proposed that directly utilizes L-Systems [PM01b]: Their L-System contains an extrusion module to generate basic building blocks. Those can then be combined using transformation operators. Geometric templates for roofs, antennae and other things add further building detail. Note that this approach does not allow creating geometric façade detail, only building shells are generated. Façade detail is added by distributing multiple textures over the shell.

3.2. Split grammars

Instant Architecture [WWSR03] was the first publication that proposed a shape grammar-based approach suitable for automatic building generation. Three main contributions were made:

- Shape and Grammar definition: A mathematical definition of Shape and Grammar in the context of computer graphics is provided. It is very similar to the one proposed for shape grammars [Sti80], described in Chapter 2.
- Split Grammar: Based on the mathematical definition, a *split* is defined as the decomposition of a basic shapes into other shapes. This split is the key to solve the *distribution* problem: It allows automatically placing shapes in relation to a parent shape.
- Production application process, production rule selection: During production application, multiple productions may match at a specific point. Therefore, at each application step, a control grammar is invoked. This grammar removes productions that do not fit into a specific position. To choose a production that is coherent with the building parameters from the remaining ones, attribute matching is performed. The production with the highest match is chosen.

Discussion Instant Architecture describes split grammars from a mathematical point of view. In my opinion, it is hard do deduce an actual implementation from such a view. The more recent publication "Procedural Modeling of Buildings" [MWH+06] provides a more pragmatic view on design grammars: Instead of mathematical definitions, an actual syntax for split commands is provided. My implementation is based on the more recent publication. Therefore I will not go into further detail of the Instant Architecture formalism, and instead provide a detailed view on the more recent publication.

3.3. Computer Generated Architecture (CGA)

In this section we will provide a detailed description on CGA proposed in [MWH+06]. CGA is able to automatically generate visually convincing building models. It has been shown to be applicable to different building styles, ranging from historical buildings to skyscrapers [MWH+06]. The main idea is to provide multiple production rules, defining different building parts. Those productions are applied sequentially, generating buildings from ground plans.

CGA consists of multiple concepts. A grouping of those concepts is provided in Figure 3.3, depicted as three columns.

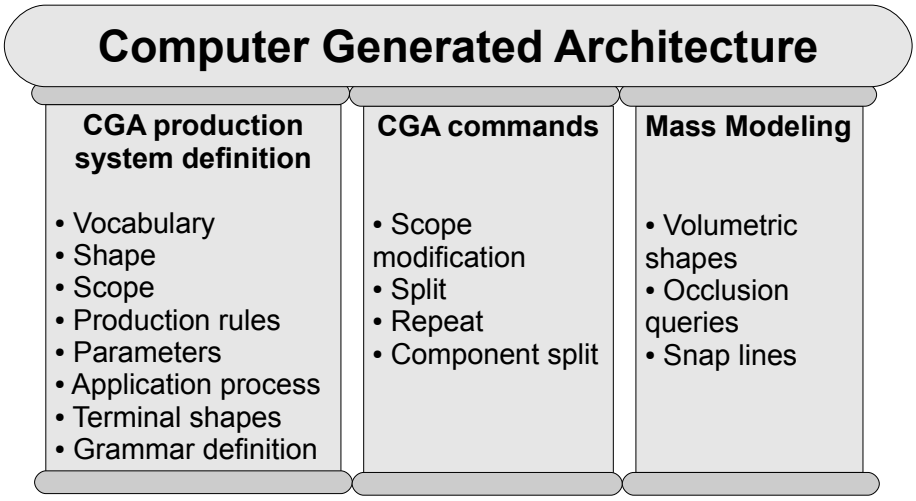

Figure 3.3.: Overview on CGA

The first column represents the most important concept: CGA is a production system. Therefore it is necessary to provide a detailed definition of this system. An overview on those definitions is provided in the column.

The second column represents CGA commands. Many functionalities of CGA are accessed with those commands. For example, the distribution of building parts is mainly conducted utilizing commands. Different commands are seen in this column.

Another important concept is mass modeling, depicted as the third column. Mass modeling allows creating complex building shells with ease. Functionalities enabling mass modeling are displayed in this column.

This section is structured according to the three columns: At first, we provide a formal definition of CGA. Then, different CGA commands along with their applications are explained in detail. Afterwards, mass modeling is explained. Finally, CGA is compared to L-Systems and other shape based approaches.

Citation Convention To avoid bloating the detailed description with dozens of references to the CGA paper [MWH$^+$06], we will use the following convention: Everything written in the sections 3.3.1 to 3.3.3 was proposed by Müller et al. [MWH$^+$06] *except* things that are written in paragraphs titled "Additional Details". All figures were created by myself, but most of them are based on figures from the CGA paper. Backus-Naur definitions were done by me. To increase readability of Backus-Naur notations, the simplification described in Appendix A is used.

3.3.1. Formal CGA Definition

A generic description of production systems was provided in Section 1.4. As CGA is a specific production system, we will now adapt this generic description to CGA. At first, we define the vocabulary:

Definition 6 *CGA operates on* shapes*. The set of symbols associated to shapes CGA operates on is called the* vocabulary U*. The vocabulary consists of terminal symbols* $\Sigma \subseteq U$ *and nonterminal symbols* $V \subseteq U$*.*

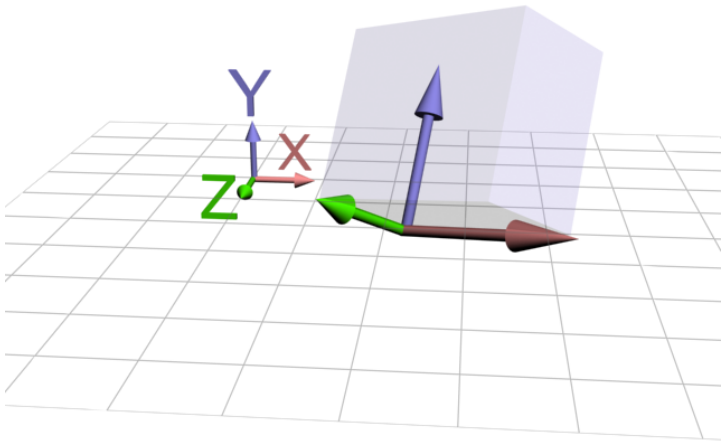

Figure 3.4.: World coordinate origin and a scope placed in world coordinates. Adapted from [MWH+06].

Müller et al. define shape in CGA the following way [MWH+06]:

Definition 7 *... a shape consists of a symbol (string), geometry (geometric attributes) and numeric attributes. Shapes are identified by their symbols, which is either a terminal symbol* $\in \Sigma$ *or a non-terminal symbol* $\in V$*. The corresponding shapes are called terminal shapes and non-terminal shapes. The most important geometric attributes are the position P, three orthogonal vectors* X*,* Y*, and* Z*, describing a coordinate system, and a size vector* S*. These attributes define an oriented bounding box in space called scope.*

Let us describe a scope in more detail: An initial scope is assigned to the axiom. During derivation, a scope is assigned to every created shape. Scopes are automatically calculated. Figure 3.4 depicts a possible scope. There is no way to directly set a scope during derivation, however, scopes can be modified with scope commands, as described later.

Commands Some functionalities of CGA are encapsulated in commands. We need to define commands in order to introduce productions:

Definition 8 *A command* $c \in C$ *is a macro that either creates new shapes or modifies properties (e.g., the scope) of existing shapes.*

The different CGA commands C are explained later in Section 3.3.2.

Please note that the term "command" was not used by Müller et al. [MWH+06]. Instead, for example the term "scope rule" was used to indicate that a scope command occurs in a specific production rule. We propose to use the term "command" in order to make the distinction between production rules and other functionalities (e.g., scope commands) more evident.

Production Rules CGA contains a list of *production rules* R in the form $v \rightarrow u$, $R \subseteq V \times (U \cup C)^+$. For easier understanding, we provide an informal syntax at first:

$predecessor(localParameters) : conditions \rightarrow successor : probability;$

The exact syntax is defined the following way:

```
<production> ::= <predecessor>
   [ "(" <symbol> { "," <symbol> } ")" ]
   [ ":" [<occlusion>] <condition> { "," <condition> } ]
   "~"
   <successor>
   [ ":"   <real>]
   ";" ;
```

<predecessor> is a string representing the symbol $\in V$ associated with the predecessor shape. <symbol> is a string defining one *local parameter* of this production. Multiple parameters can be defined using a comma separated list. <condition> defines one boolean condition that must evaluate to true if the production should be applied. <occlusion> is a specific kind of boolean condition, used for occlusion queries, as described later. Using a comma separated list, multiple conditions can be defined. <successor> is a list of shape symbols $\in U$ and commands $\in C$. During derivation, commands are executed and the predecessor is replaced with the successor. The optional <real> number describes the probability $prob$ of this production to get selected. When no probability is provided, $prob$ is initialized to 0.5.

A production may look like this:

```
fence(height) ~ S(1,height,0.01a) Repeat(X,2)
   { I(CUBE,"cat:balcony",t,1,t) };
```

This creates a fence as seen in the middle of Figure 3.8. `height` is a local parameter, and can be accessed in this production. Here, the parameter is used in the scaling command `S`. When `fence` is utilized, this parameter must be defined, as seen in the following code:

```
A ~ fence(10.0);
```

Local and Global Parameters: In the previous paragraph we described how to define and use local parameters. They can only be accessed in the production they are defined in. Contrary, the scope of global parameters extends to all productions, and are defined the following way:

```
const parameterName = 10.0f;
```

When a global parameter is defined, any production can directly use it, for example:

```
fence ~ S(1,parameterName,0.01a) Repeat(X,2)
   { I(CUBE,"cat:balcony",t,1,t) };
```

Priorities and Probabilities When introducing production rules, we said that a probability *prob* can be assigned to each production. This probability is used to decide which production to use when multiple productions match a specific shape during derivation. The derivation, or production application process, is explained in the next paragraph.

Priorities are another CGA concept. Priorities are used to determine the order of production applications. This can be used to derivate the building to a specific detail level.

A priority is assigned to every production rule, using the following syntax:

```
priority <uint> ":"
```

The unsigned integer <uint> is then assigned as priority value to every subsequent production. For example (The successor is not important for now, therefore we just write "..."):

```
priority 1:
winWall ~ ... ;
wall ~  ... ;
priority 2:
window ~ ... ;
```

Here, the productions with predecessor `winWall` and `wall` have the priority 1 assigned, while the last production has the priority value 2.

Production Application Process Now that we have the vocabulary U and production rules R defined, we can define the production process. The general idea is as follows: New shapes are produced by starting with an initial shape, and repeatedly applying productions. Müller et al. [MWH+06] formalized the basic production process the following way:

Definition 9 *A configuration is a finite set of basic shapes. The production process can start with an arbitrary configuration of shapes A, called the axiom I, and proceeds as follows: (1) Select an active shape with symbol $B \in V$ in the set (2) choose a production rule with B on the left hand side to compute a successor for B, a new set of shapes $BNEW$ (3) mark the shape B as inactive and add the shapes $BNEW$ to the configuration and continue with step (1). When the configuration contains no more non-terminals, the production process terminates.*

This process is extended to utilize the priorities assigned to each production: In step (1) the shape with the production of highest priority is chosen. In step (2) the production rule probabilities are used to decide which production to use when multiple possibilities exist. Therefore, in our implementation a realization x_i of the uniformly distributed stochastic variable $X \sim U_{0,1}$ is assigned to every possible production R_i. x_i is then multiplied with the corresponding production probability $prob_i$. The production with the highest result of $x_i \cdot prob_i$ is then chosen for further derivation.

Additional Details: In our implementation, we use a priority queue for the production process. This is slightly different compared to the previously described process, and works as follows:

Every priority queue entry stores two things: The production that should be applied and a pointer pos to the position in the configuration where the successor should be inserted.

To actually define the axiom I, we select a specific production P. The successor of this production represents the axiom. This may seem unintuitive at first, however this method defining an axiom has the following advantage: We can uniformly handle the axiom shape creation and production application in our implementation. It is not necessary to write specific code to create an axiom shape, because the axiom shape is created automatically during the first iteration.

Before we start the process, we clear the configuration. The selected production P is inserted into the priority queue, with pos set to the first configuration position. Then the production application process is started:

1. The production on top of the queue is selected as the current production. The shape on the position pos of the top queue entry is our active shape B.
2. Every command (e.g., split or turtle command) in the current production's successor is executed, resulting in a new set of shapes $BNEW$. Different types of commands are explained later.
3. For every shape $S \in U$ occurring in $BNEW$, we search for a production that has this shape's symbol as predecessor. When multiple matches occur, we select one production utilizing production probabilities, as described earlier. The selected productions are inserted into the priority queue, along with the position of the occurrence of S. The queue is sorted according to the production priorities.
4. Now the actual production application is performed: We replace shape B in the current configuration with $BNEW$. However, the old shape is not deleted entirely, instead it is marked as inactive. Internally, a hierarchy of production applications is built to enable queries later on.
5. The current production is removed from the queue.

6. Step 1 is performed until the queue is empty, or no production has a priority above a user-defined threshold.

Note that we start with an empty configuration, using the described approach the configuration is automatically set to the axiom during the first iteration in step 4.

Please consider the following example. Ignore the commands `Subdiv` and `I` for now, they are explained later. The only important things are the `priority` commands, as they set the priority of the following productions:

```
priority 1:
winWall ~ Subdiv(X,1,1,1){ wall | window | wall };
wall ~  I(PLANE;"wall.jpg");
priority 2:
window ~ I(PLANE;"window.jpg");
```

When the production with the predecessor `winWall` is in the queue, the further derivation creates queue states as seen in Figure 3.5.

The queue based approach has the following advantages over the interleaved grammar/control grammar approach:

- The application order of productions is defined, therefore a control grammar is not necessary.
- The building can be generated until a specific detail level is reached. This can increase the performance when working with large models.

Terminal Shapes We have not yet explained terminal shapes: These shapes $\in T$ have a mesh geometry and texture associated, and have no successor for further derivations. Please note that it is not possible to directly define terminal shapes in our implementation. Instead, a command to insert the mesh with the texture into the current derivation is provided, implicitly creating a terminal shape. When such a command occurs the contained geometry is fitted to the current scope. The syntax is as follows:

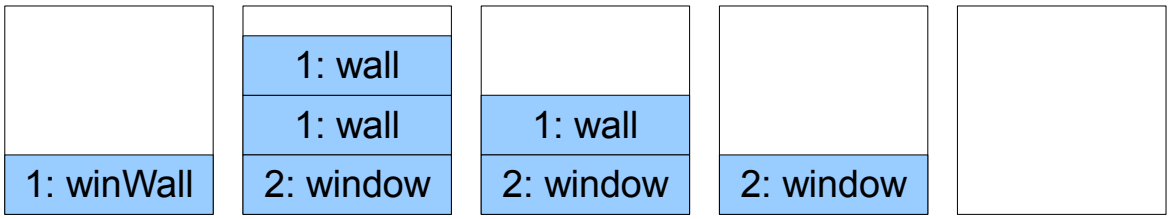

Figure 3.5.: From left to right: Queue states after application of production with predecessor `winWall`. The insert positions are not displayed.

```
"I" "("
  ( CUBE | PLANE | CYLINDER | SPHERE  | '"'<filename>'"' )
  "," '"' <filename> '"' ")";
```

The first argument defines the type of the shape. Standard primitives as well as generic meshes described by `'"'<filename>'"'` are possible. The second `'"'<filename>'"'` parameter determines the texture to use. For example:

```
I(CUBE, "wall.jpg")
```

This inserts a cube fitting the current scope, with the texture `"wall.jpg"` applied.

CGA Grammar Definition We have introduced the vocabulary U with terminal symbols Σ and non-terminal symbols V. Further, productions R and the axiom I were defined. Now it is possible to define the CGA grammar:

Definition 10 *The tuple* (V, Σ, R, I) *is called a CGA* grammar.

This concludes our formal CGA introduction. The following sections will explain some aspects of CGA in greater detail.

3.3.2. CGA commands

When formally introducing CGA, commands where mentioned a few times. Here we will describe the different possible CGA commands, and point out their applications.

Scope Commands L-Systems have commands for scope manipulation. Those commands are applied to CGA: `T(tx,ty,tz)` translates the scope position along the object-space orientation of the scope. `R(axisX,axisY,axisZ,angle)` rotates around a specific axis. `RX(angle)`, `RY(angle)` and `RZ(angle)` rotate around the specified object-space axes of the current scope. `S(sizeX,sizeY a,sizeZ)` resizes a scope to the specified size. Scope commands allow flexible placement of shapes, and are thus useful for complex building generation.

Additional details: The size parameters can either be multiplicative to the current size, or absolute values, indicated with a succeeding `a`.

Split Command A split command divides the current scope into multiple parts, along a specified axis. For example, multiple floors can be generated this way. A shape or command is assigned to each part. The syntax, described using Backus-Naur form, is as follows:

```
<subdiv> ::= Subdiv "(" <axes> ";"
   <subdiv_size> [{ "," <subdiv_size> }]  ")"
   "{" node [{ "|" <node> }] "}";

<subdiv_size> ::=  <floatfield> ["r"];

<axes> ::= [X] [Y] [Z];
```

For example, this is a valid split command:

```
Subdiv(Y; 1, 2r){floor | groundfloor}
```

`Y` describes the axis that is split into multiple parts. Here, a split along the vertical axis is performed. "`1, 2r`" defines the sizes of the resulting scopes. `1` represents an absolute size, while `2r` denotes a relative size. Relative sizes may result in non-uniform scaling and should therefore not be used for architectural parts that need a constant aspect ratio. Relative sizes are calculated as follows: $size = rel_i * (SplitAxisSize - \sum abs_i) / \sum rel_i$, where rel_i is a specific relative size value, and abs_i represents an absolute value. "`floor | groundfloor`" determines which shapes should be placed in the resulting scopes.

Figure 3.6 shows the result of this split, provided the initial scope size was (4,4,0).

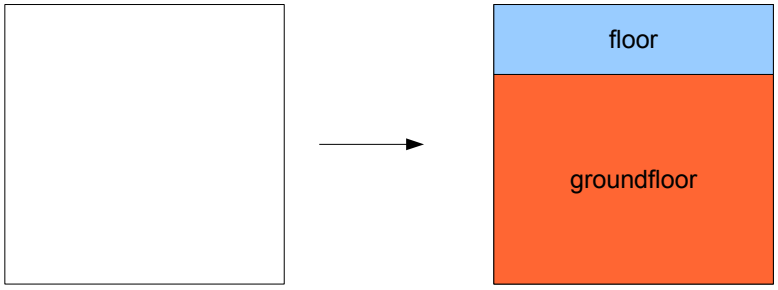

Figure 3.6.: Result of split operation

Repeat command Contrary to a split command, a repeat command has only one shape or command associated. This shape or command is repeated along a specified axis as often as possible for a given repeat width. For example, it is easy to place multiple windows on floors when utilizing this command. The syntax is as follows:

```
<repeat> ::= Repeat "(" <axes> "," <float> ")" "{" <node> "}";
```

An example repeat command is:

```
Repeat(X, 2){window}
```

This results in the shape `window` to be repeated along the `X` axis with the size `2` as long as it fits the current scope.

Additional details: When the repeat size is larger than the axis size, it is clamped to the axis size. In case the current scope is no exact multiply of the repeat size, the repeat size is decreased to the nearest possible size that allows an exact fit of repeated shapes. The new repeat size is calculated with the following formula:

$$newSize = RepeatAxisSize / \lfloor RepeatAxisSize / oldSize \rfloor$$

Figure 3.7 (a) shows the result of this repeat, provided the initial scope size was (6,2,0). Figure 3.7 (b) demonstrates how repeat sizes are adjusted to get exact fits to the scope, in case an exact multiply does not fit the scope.

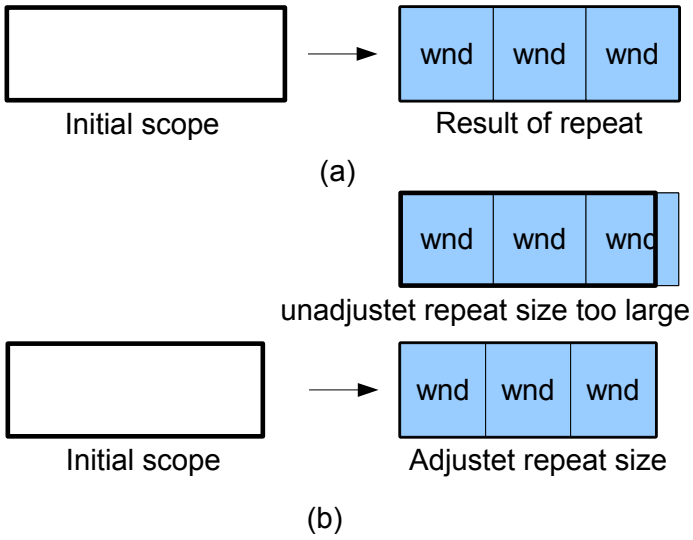

Figure 3.7.: (a) Result of repeat operation (b) Adjustment of repeat sizes

Component Split Command This command allows decreasing the dimensionality of the current scope, and thus working with fewer dimensions. A subsequent scaling command can increase the number of dimensions again. At first, the possible applications of this command may not be obvious. We will describe possible applications later, let us define the syntax at first. A slightly modified version of the component split of the one proposed [MWH+06], is defined as follows:

```
Comp "(" <objType> "," <objPos> "," [<uint> { "," <uint> }] ")"
```

```
          "{" <node>  "}" ;
<objType> ::= FACES | EDGES | VERTICES ;
<objPos> ::= ALL | TOP | BOTTOM | SIDE ;
```

<objType> describes to which dimension to split. FACES are 2D, EDGES are 1D and VERTICES are 0D and thus represent a point in space. <objPos> defines which geometric primitives from the current volumetric shape to use for the split. Those primitives can be reduced further by using the optional <uint> numeric parameter: Only primitives with a sequential number that occurs in this list are used. <node> determines the CGA shape to place or command to execute on each split location.

Let us now provide some examples to explain where component splits are applicable. By splitting to the side edges of a façade we can place bricks on façade edges, as seen in Figure 3.8 on the left side:

```
Comp(EDGES, SIDE){corner}
```

Adding a fence to a planar roof is possible by a component split to the top edges of the roof, as depicted in Figure 3.8 middle.

```
Comp(EDGES, TOP){fence}
```

A vertex split can be used to place an shape on a specific building vertex, for example a tower can be added:

```
Comp(VERTICES, TOP, 1){tower}
```

Note that ", 1" means that the tower is only placed on the first occurring vertex. Figure 3.8 right shows the result.

Until now, no example for a face split was provided. There will be one in the next section.

Figure 3.8.: Component splits. From left to right: Side-edge split for corner, top-edge split for fence, vertex split for tower

Additional details: As the dimensionality is reduced, some orientation axes are not used anymore. However, as a later scaling operation increases the dimensionality again, those axes must still be initialized to a meaningful value. For splits to 2D this is simple: The face normal is used for the Z axis. For edges, things are more complicated, as two orientation vectors are undefined. Therefore, we define the following behavior, visualized in Figure 3.9:

- Top and Bottom Edges: Those splits are primarily used to place fences, as seen in Figure 3.8 middle. To get the scope aligned for easy fence placement, the following calculations are used: The X axis is assigned to the edge vector. For the Y axis, the top/bottom face normal is used. The cross product of X and Y yields the Z axis.
- Side Edges: In order to easily place corners, as seen in Figure 3.8 left, the Y axis is assigned to the edge vector. The Z axis is calculated as the average of the two adjacent side face normals. The cross product of Y and Z yields the X axis.

For vertex splits, no axis is defined. Therefore we define the Y axis to be the adjacent top or bottom face normal. The Z axis is the average of the two adjacent side face normals. The cross product of Y axis and Z axis results in the X axis.

Workflow using Component Split Using face component splits, a new workflow to get from ground plan definition to individual façades is possible. It is depicted in Figure 3.10: A 2D ground plan is provided. Using a scale command yields a 3D shape. Now the component

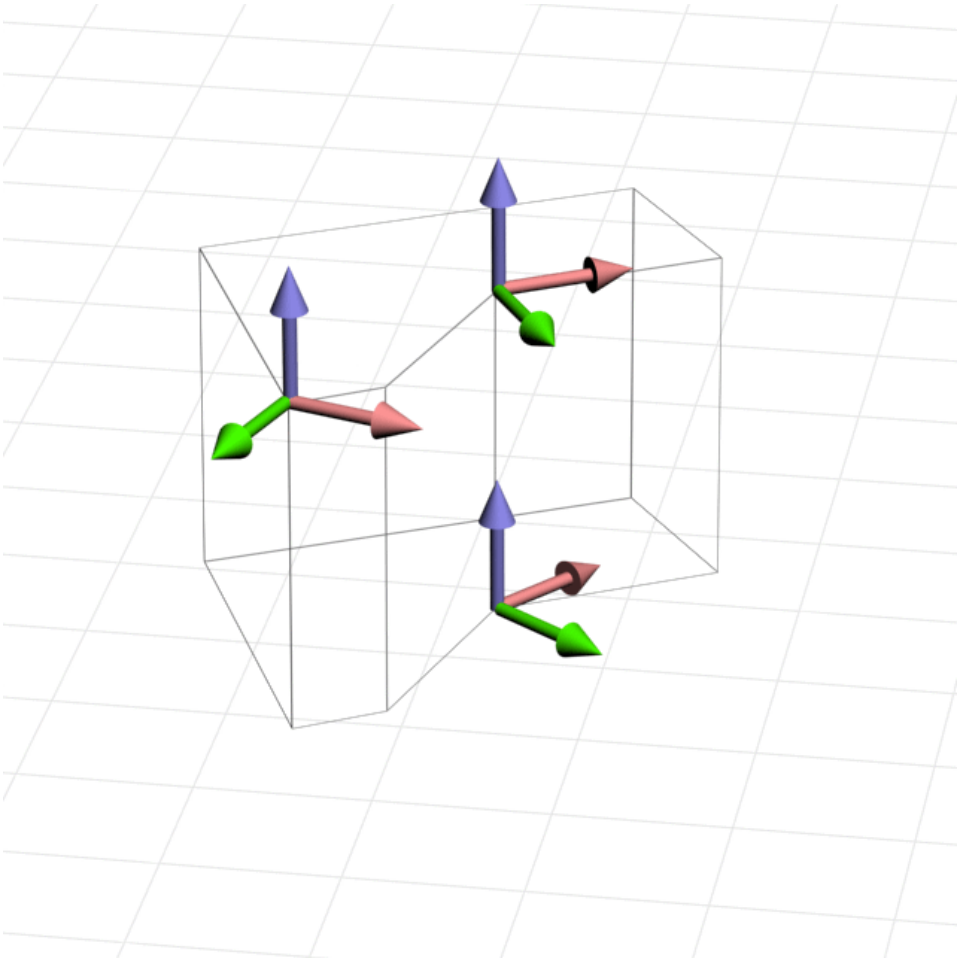

Figure 3.9.: Alignment of scopes after component splits. Left: Result of a top-vertex split. Top right: Result of a top-edge split. Bottom right: Result of a side-edge split.

split is performed, creating aligned 2D scopes for every façade polygon. Every façade can now be easily modeled in 2D, or extruded back to 3D.

This workflow is achieved with the following code:

```
axiom ~ S(1,houseHeight a, 1) Comp(FACE,SIDE){facade);
facade ~ S(1, 1, wallThickness a) ... ;
```

Please note that absolute scalings are used to get to 3D, as all relative scalings would result in zero size.

3.3.3. Mass modeling

In terms of the possible building variety, mass modeling is the most important contribution from [MWH+06]. The main idea is simple: By using basic volumetric shapes which can be placed freely using scope commands the building shell is very flexible.

However, two problems emerge when using this approach: At first, overlapping volumetric shapes may result in unwanted intersections of derived shapes. For example, windows may intersect with walls. This is avoided by using *occlusion queries*: Before windows are placed, a check for intersections with walls is performed.

Another problem is that shapes on different volumetric shapes may not line up coherently. Therefore, in order to align shapes on different base shapes, *snap lines* are employed. Split and repeat positions are aligned to those lines to create coherent buildings. We will describe those concepts in greater detail in the remainder of this section.

Occlusion Queries Figure 3.11 demonstrates the occlusion problem: Windows on a different volumetric shape may intersect other windows or walls. The solution is to check each window for an intersection. Only when no intersection occurs, the window is drawn, otherwise a wall is drawn. For the actual intersection test, an octree structure was proposed, and calculations were done on the CPU [MWH+06]. To accelerate the queries we propose to use GPU accelerated occlusion queries, as described in Section 5.2.1.

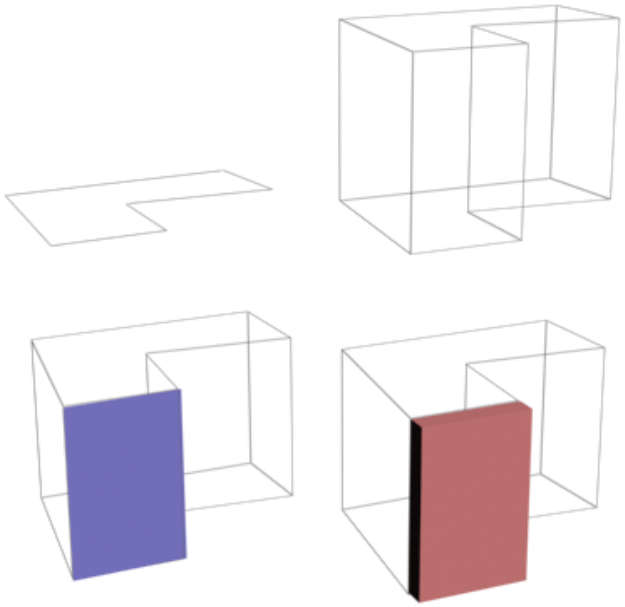

Figure 3.10.: Workflow using component splits. Top left: Start ground plan. Top right: extruded ground plan. Bottom left: side-face component split yields scopes for façades (only one façade is drawn here). Bottom left: Façade is scaled back to 3D.

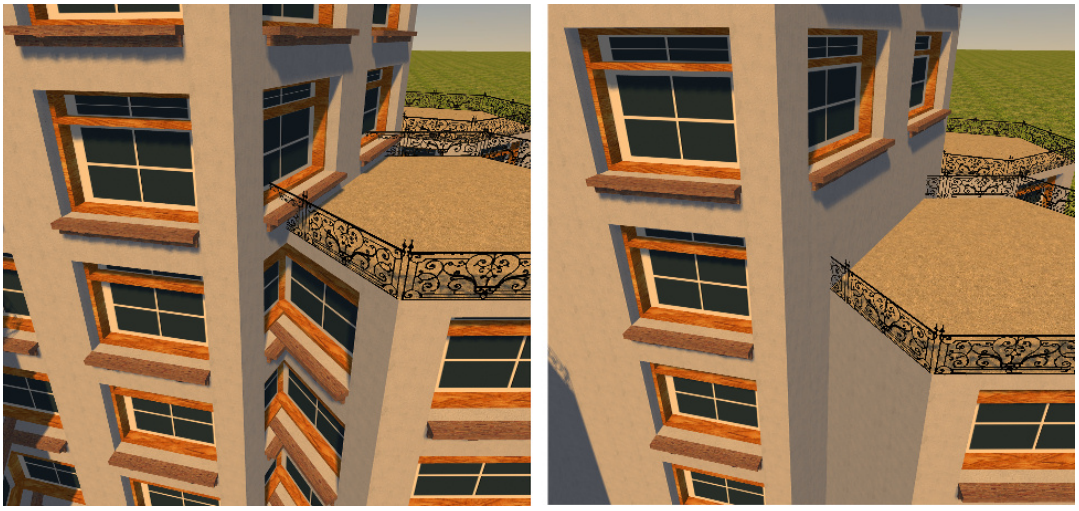

Figure 3.11.: Left: Without occlusion queries, windows intersect with walls. Right: With occlusion queries.

It is difficult for a program to decide on which productions to perform occlusion queries. Therefore the user has to explicitly tell the program to use those queries on specific productions. The syntax is as follows:

```
<occlusion> ::= ( Shape | Scope ) ".occ" "(" ( noparent | all ) ")"
                "==" ( none | part | full )
```

It is used as a boolean test for productions, for example:

```
window: Scope.occ(noparent) == none ~ fullWindow;
window: Scope.occ(noparent) == part ~ wall;
```

When `Scope` is used, intersection tests are performed on scope level, with `Shape` the actual geometry is used for tests. `all` tests every existing shape for intersections, `noparent` ignores the current shapes's parent. This is often used, as by definition of the split command, every child is contained in the father's scope, and thus always intersects with the father.

It would be an interesting future work to extend this approach in order to achieve more fine-grained results: For example, instead of just replacing the window with a wall, we could resize it, or replace it with a smaller window. This functionality is *not* presented in [MWH+06], it would be interesting to add it to our implementation.

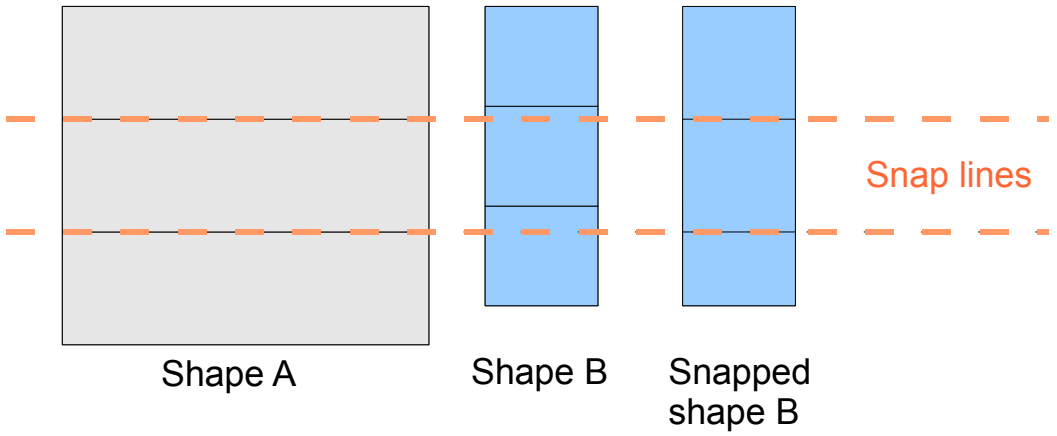

Figure 3.12.: Application of snap lines.

Snap Lines When a user inserts the keyword `Snap`, snap lines are generated. All subsequent split and repeat operations align their division lines to those snap lines. This way, coherency across multiple volumetric shapes can be achieved, as displayed in Figure 3.12. Note that our implementation does not support snap lines at the moment, therefore a detailed syntax is not provided.

3.3.4. Relation to L-Systems

The main similarity is the concept of a production process based on a grammar, replacing objects with other objects. Additionally, the turtle commands are used for scope manipulations. However, there are some differences:

- L-Systems operate on strings: A substring is replaced with another string. CGA operates on shapes: A shapes is replaced with another shape.
- CGA uses serial derivation instead of parallel derivation.
- The result of a L-System derivation is a string. This string does not directly represent geometry, it has to be interpreted first. It is not possible to directly create geometry, as

we do not have the scope information necessary to place geometry during derivation. Parallel derivation makes scope information gathering during derivation difficult.

In CGA, we need the scope information during derivation to calculate the outcome of split commands. As serial derivation is employed, gathering scope information during derivation is possible. Using this information, we can directly create the geometry when a terminal shape occurs. Thus, there is no need to interpret the derivation tree afterwards.

- L-Systems simulate growth-like processes in unrestricted space, while CGA describes a series of partition steps filling a predefined space [WWSR03]. The partitioning steps are performed using split and repeat commands. Of course, as CGA also has scope modification commands, growth like processes can be simulated too. However, this is not the main goal of CGA as architecture can be better represented using partition steps in most cases [WWSR03].

3.3.5. Comparison to Other Shape-Based Approaches

CGA, Instant Architecture, shape grammars and set grammars are all shape-based production systems. Here we will point out the main differences of those systems.

Set grammars are a simplified version of shape grammars, as described in Section 2.2. As CGA and Instant Architecture are based on set grammars, we will *not* include shape grammars in our comparison.

Instant Architecture can be seen as a set grammar extended with split commands. Further, a production application process utilizing control grammars is defined. CGA extends Instant Architecture with scope, repeat, and component-split commands. Additionally, a priority-based production application process is used, and the concept of mass modeling is introduced.

The main difference between CGA and set grammars is the scope definition, and commands using this scope information. This allows easy distribution of shapes on façades.

Applicability CGA and Instant Architecture are designed to create visually convincing building models, instead of accurate floor plans. To my knowledge there is no set grammar approach that is able to create such models. This raises the following questions: Why is there no set grammar based approach? Is it because set grammars are not able to create such models?

In my opinion, set grammars are indeed not applicable to detailed façade generation. The main reason for this is the lack of a split and repeat command. Without those commands it is complicated to distribute shapes on a façade. While it would be possible to define a split command using the set-grammar syntax, we do not know how a repeat command or component-split command could be represented in this syntax.

CGA versus Instant Architecture CGA is based on split grammars. However, several major contributions set CGA apart from split grammars. The following list highlights all contributions that set CGA apart from Instant Architecture:

- Instead of interleaved grammar/control grammar application, a priority-based production process is employed. No control grammar is used.
- They are the first to actually *define the syntax* of split commands. This greatly helps when doing a computer implementation, as a concise notation is provided. Please note that the syntax is described in a way that is understandable for a human reader, however, a precise syntactic definition suitable for a parser is not provided. We therefore contribute an exact syntactic definition using Backus-Naur form in Appendix A.
- A new type of split command is defined: The *component split*, allowing to reduce the dimensionality of the current scope.
- By employing the component split, a new *workflow* to get from a ground plan to façades is possible.
- *Mass modeling* using volumetric shapes and occlusion queries allows to create more complex buildings.

Let us discuss the implications of those contributions: Mass modeling and scope commands greatly enhance the expressivity of CGA, as complex building shells are possible. Such shells

would be complicated to model in Instant Architecture, as multiple splits would be necessary to achieve complex shapes.

Further, the component splits simplify the placement of shapes on geometric primitives of other shapes. This makes distribution of shapes easier, and thus simplifies building generation.

The priority-based production process is advantageous because we can derivate to a specific detail level. As no control grammar is used, the grammar creation is easier. However, the lack of a control grammar reduces the flexibility, as a separate control grammar is able to encode design decisions more flexibly. This is because we have two layers of grammars. Therefore, we have a trade-off between simpler grammar creation and more flexibility.

To sum this up, in my opinion CGA is better suited for building generation, but the flexible control grammar of Instant Architecture may be advantageous in some situations. Therefore it would be interesting to explore the applicability of a control grammar to CGA.

3.4. Other Related Work

In this section, previous work not directly related to shape grammars, but nevertheless sharing some common points, is reviewed. Ideas that are applicable to CGA are pointed out.

Groundwork Templates Larive and Gaildrat propose to use a restricted version of split grammars for easier façade generation [LG06]: Only five productions that can be combined in different ways are provided. As this approach greatly reduces the flexibility of split grammars (split grammars allow an unlimited amount of productions), we do not think it is a practicable solution for more complex buildings. However, their paper provides some interesting ideas: In order to cope with non-horizontal building footprints, a building is automatically extruded with a groundwork to fit the footprint, as show in Figure 3.13. Larive and Gaildrat name this functionality *groundwork templates*. It would be interesting to integrate this functionality into CGA.

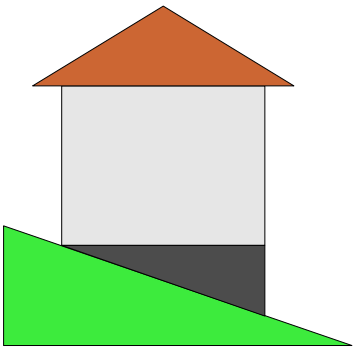

Figure 3.13.: Automatically extruded groundwork, called groundwork templates. Adapted from [LG06]

CityGML Instead of flexible production rules, a rigid class hierarchy of objects is defined [KGP05] [DB05]. For example, walls, columns and wall sections are defined. Those objects can then be stitched together. In my opinion, this approach is suitable for simple buildings, but can not handle more complex ones. However, one interesting method is proposed: Instead of defining just one ground plan for a building, every floor can have a floor plan. Those floor plans are based on prototypes: Only parts different from the ground plan are specified. This way, protrusions like bay windows can be modeled very intuitively. CGA would probably benefit from this functionality.

Roof Generation While a generated roof can be easily incorporated into CGA, by using CGA productions on the roof scopes, the actual generation of roofs is not possible using CGA syntax. Müller et al. propose the following syntax to describe roofs [MWH+06] in CGA:

```
Roof "(" '"' ( hipped | gambrel | cone | gabled | mansard) '"'
          "," <float> ")" { <node> }
```

The first parameter specifies the roof type. The second indicates a roof angle, while `<node>` is the production or command that is called for every roof scope. Please note that this just describes parameters of a roof, how to actually generate the roof is *not* specified in CGA. A detailed roof generation algorithm, based on straight skeleton generation, is presented in [LD03]. This algorithm could be integrated seamlessly into CGA.

3.5. Summary

At first, we presented the first grammar based approach used in computer graphics. Then the first shape grammar based approach, called split grammar, was introduced. A detailed look on CGA, the paper our implementation is based upon, was provided. We added some details that were not present in the previous publication, and provided an exact Backus-Naur form for some commands. In Appendix A, a full specification of the CGA language is provided. This helps in creating a correct parser. Further, the implications and possibilities of the component split were explained. Finally, other work targeted at building generation was presented.

When looking at the Section 1.5, it is visible that CGA is not perfect, and therefore we will propose improvements in the next chapter.

Chapter 4.

Interactive Computer-Generated Architecture

As pointed out in Section 1.5, CGA has some drawbacks. This chapter will provide solutions to alleviate some of them.

At first, our implementation of an interactive editor is explained. This targets the usability problem. Then, language enhancements are proposed to remove some ambiguities of CGA. The performance problem is the subject of the next chapter.

4.1. Interactive Editor

The main modeling paradigm introduced in [MWH+06] is based on editing text files. In Instant Architecture [WWSR03], the following user interaction levels with shape grammars are defined:

1. Modify production rules
2. Modify start shape parameters
3. Modify parameters of other shapes

Following the CGA paradigm [MWH+06], all of those interaction tasks require a change in corresponding text files. Our goal is to provide graphical interaction for at least some of those areas. To further evaluate possibilities of user interaction, let us describe the user tasks in more detail, by describing what a user may want to achieve with a specific modification:

1. Modify production rules
    - Create completely new production rules. Also create a new successor shape.
    - Manually steer the derivation process. This defines which production rule to use when there are multiple possibilities during derivation.
2. Modify start shape parameters
    - Adjust global building parameters
3. Modify parameters of other shapes
    - Fine-grained control over building parameters

While the previous approach requires textual changes for all of those interactions, our method allows graphical interaction for all of those points except "Create new production rules". How this is achieved will be explained in this section. Therefore, at first an overview on the interaction techniques is provided, focusing on how a user actually performs modifications. Then the focus is laid on the underlying concepts used to achieve this functionality.

4.1.1. Overview on Graphical Interaction

New Production Rules To create new production rules with a new shape as a successor, the CGA code must be modified textually, just like with the old approach. Figure 4.1 shows the syntax highlighting editor used for this task. The user can sort CGA files into folders using drag-and-drop. Those folders are only used to tidy things up, they do not have any effect, as the different CGA files are internally handled as one big file.

After parsing of the files, all encountered productions are displayed in a tree view. This tree view provides an overview of all production rules. Further, it is possible to steer the derivation process by dragging elements from this tree view, as described in the next paragraph. We

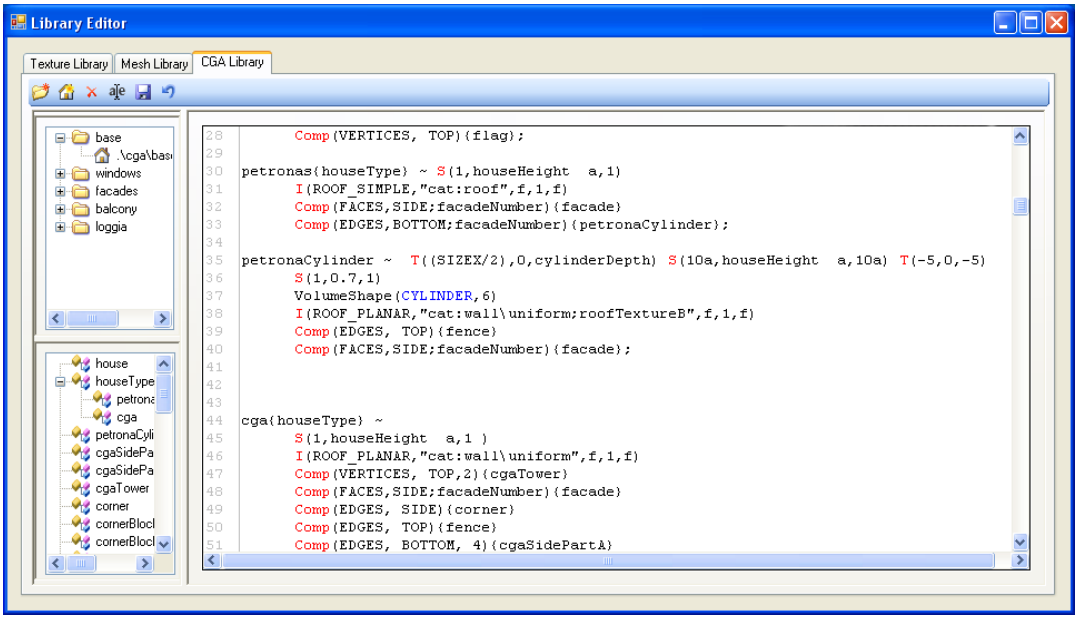

Figure 4.1.: The syntax highlighting text editor is shown on the right. On the left side, CGA files are sorted into folders. Below that, the hierarchy of all parsed productions is shown.

call this tree view the CGA library. As you may notice, the tree view is hierarchical, for example the production with the predecessor `houseType` has two child productions. How we determine those hierarchies is subject to Section 4.2.2.

Steering the Derivation Process CGA allows multiple production rules to have the same predecessor. When such a production is applied, a specific production from those multiple matches has to be selected. This enables flexible shape combinations, as multiple derivation variants are possible.

The program can automatically choose a production, for example by selecting a random one. This way, multiple variants are possible without changing the CGA code.

However, when the user wants to explicitly specify the production to be used, changes to the CGA code are necessary. One method is to explicitly name the production to be used by changing the production rule, as seen in the following example:

```
balcony ~ ... fence ...;
```

```
//different fence possibilities:
fence ~ solidFence;
fence ~ wireFence;

//changed to:
balcony ~ ...  solidFence ...;
```

Here, the user changed from a generic "fence" production to the specific "solidFence". Another possibility is introducing a parameter to fence, or using a control grammar-like approach, as done in Instant Architecture. All those methods have one thing in common: When a user wants to explicitly specify a production from multiple choices, manually editing a text file is necessary.

This is where the new approach comes into play: It allows dragging the fence type from the CGA library and drop it on the old fence. The building is immediately rendered again to reflect the changes. Figures 4.2 and 4.3 illustrate this process. At first, the balcony is selected. Then, `fenceSolid` is dragged on the balcony, resulting in an immediate new rendering as seen Figure 4.3. No manual text file editing is necessary.

Of course, the information which production to use during derivation still has to be saved somewhere into a text file. However, our approach does this automatically, no editing by the user is necessary. Where this information is saved is explained in the next section.

Modify Parameters Local and global parameters were described in section 3.3.1. The visual modification of global and local parameters is unified into a single approach: Sliders and numeric text fields are automatically created for every parameter.

For global parameters, this is trivial: At first, the application creates a list of all productions occurring in the current building. Then those productions are searched for usage of global parameters. A slider is created for every used parameter, and is initialized with the current value.

However, the handling of local parameters is not trivial, as one production rule with local parameters may occur multiple times in the building. There are two ways to handle this:

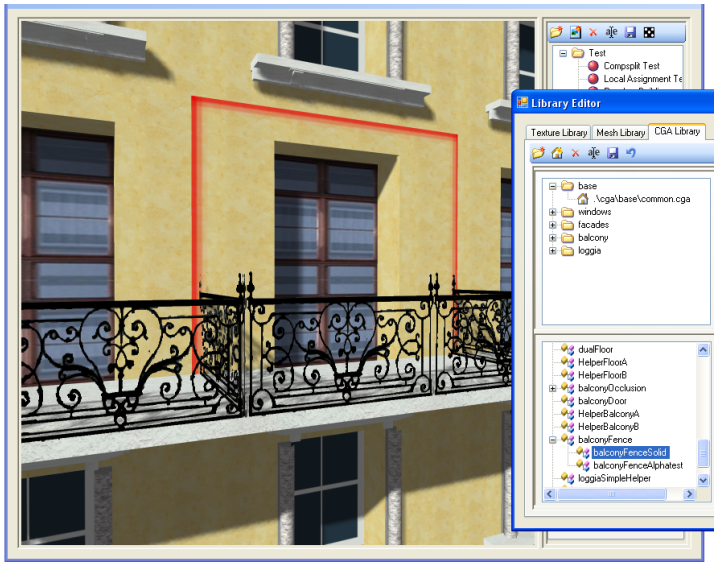

Figure 4.2.: Start of a drag-and-drop operation

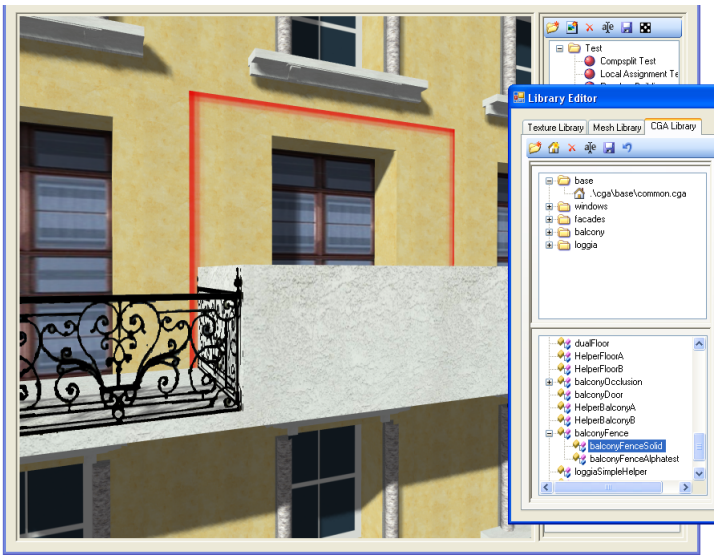

Figure 4.3.: Result of the drag-and-drop operation

- Create a slider for every occurrence of the local parameter: While this provides full control, the user interface would be overloaded. Further the user can not directly see which slider belongs to a specific production instance.
- Create only one slider: Here, two problems arise: At first, it is not clear to which value the slider should be initialized, as the multiple occurrences of the productions may have different parameter values. The more serious problem is that we lose local control, as one slider changes the parameter of multiple production instances.

Both solutions are not satisfactory. Therefore we use another approach: We do *not* map local parameters to sliders. Instead, we introduce a new parameter type, called *subtree parameter*. This parameter type will be explained in Section 4.2.1. For now, only one property of subtree parameters is important:

Subtree parameters can be defined on multiple levels. For example, a parameter value can be set for the whole house, and also for individual floors. How to actually define on which level a parameter should be set can be specified with shape picking, as described in the next paragraph.

When utilizing multiple level definitions, one slider for multiple occurrences is possible: To get an initialization value, we just query the value of the parameter on the current level. When the parameter is not defined on this level (or any higher level), the default value is used. Local control is achieved by additionally setting the value on lower levels. For example, we can set the parameter for each floor individually. When the parameter value is defined on multiple levels, the definition on the lowest possible level is used for a specific production rule. We can see that both disadvantages of just using one slider are resolved, therefore we can use this option for subtree variables.

Finally, when we have sliders for global and subtree parameters, these parameters can be modified using the generated user interface elements. A change of parameters immediately updates the rendered building. In the bottom right of Figure 4.4, the generated GUI elements can be seen. Figures 4.5 and Figure 4.6 show how parameters are changed: The user drags the slider of `balconyDepth`, and the building updates instantly.

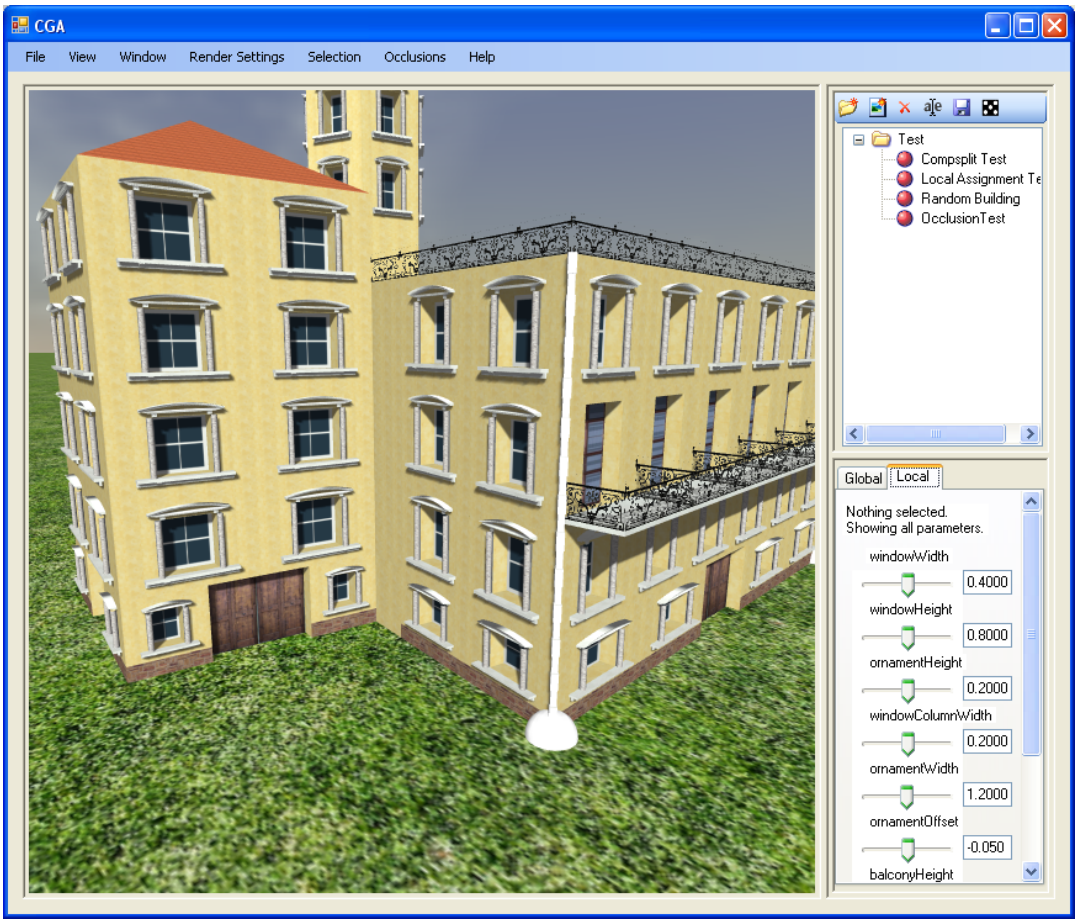

Figure 4.4.: Main user interface, showing parameter modification possibilities.

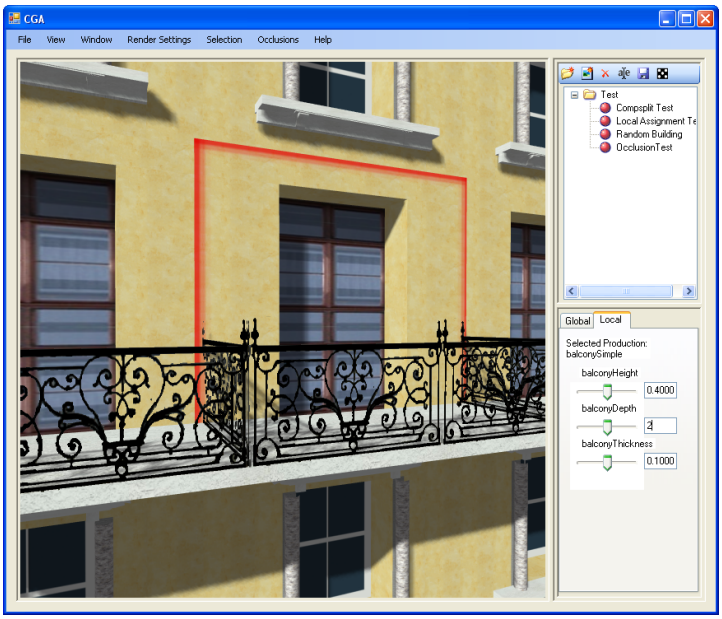

Figure 4.5.: Before parameter balconyDepth is modified

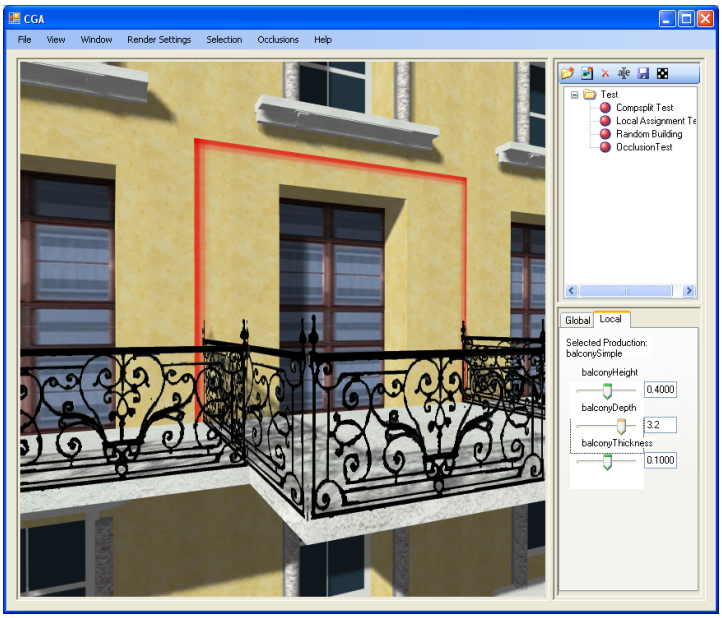

Figure 4.6.: The dragging of slider balconyDepth modifies the balcony.

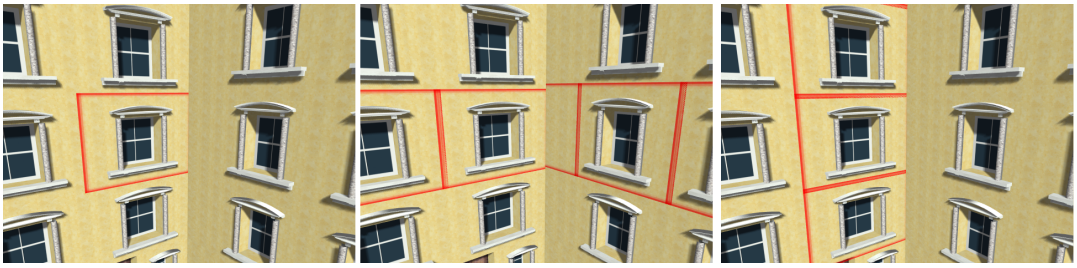

Figure 4.7.: Single/Row/Column picking modes

Fine-Grained Control In order to enable local changes of subtree parameters, the area of effect must be determined at first. Therefore, shape picking is employed. Picking is performed with a right mouse button click. All terminal shapes are then searched for intersection. The user can select single shapes, whole rows, whole columns and façades. As only terminal shapes are directly selectable, the user can issue a menu command to select the father shape of the current shape. Figure 4.7 illustrates the different picking modes.

When a shape is picked, all contained subtree parameters are shown in the GUI, and the user can change those parameters. The value of each parameter is queried for the selected level and initialized with this value. Any changes of the parameters are saved for this specific level, thus full local control is provided. Where to actually save those values is described in the next section.

Note that this method allows even finer control than the text editing approach: Different parts of the building can have different subtree parameter values, without changing the CGA code.

To achieve this with the old approach, every production that needs different parameters on different building parts needs to be included multiple times in higher level productions. Another possibility is to create control grammars to distribute those parameters. Both approaches require significant changes to the CGA code, and are thus not as practicable as direct GUI manipulations.

4.1.2. Underlying concepts

While the previous section provided an overview on user interaction, this section will focus on the underlying concepts that are used to achieve those interaction possibilities.

Applicability of Drag-and-Drop When a user drops an element, it is not trivial to determine where to actually apply this element. For example, if the user has selected a whole floor and drops a window texture, this texture should only be applied to windows and not to every textured shape in the selection.

To solve this problem, textures, meshes and productions are sorted into *hierarchical categories*. For textures and meshes, those categories can be visually created using the texture or mesh library editor, as seen in Figure 4.8. The mesh library is conceptually the same as the texture library, therefore we only show the texture library.

The hierarchy of productions can not be specified graphically, instead it has to be provided explicitly in the CGA code. This is explained in Section 4.2.2.

Now that we have hierarchical categories, we can implement the following functionality: When an element is dropped, the category of the element is compared with the categories of each selected element. If the categories match or the dropped element's category is a subcategory of the selected shape, the element is replaced with the new one.

Definition of Local Positions The position of a local change must be saved somehow. Therefore we employ the spatial locator introduced by Wonka et al [WWSR03]. Instant Architecture uses this locator for the control grammar and defines it the following way [WWSR03]:

> The spatial locator c depends somewhat on the type of split rule employed, but can usually be specified by a row-, column- and layer number (for 3D splits), including special markers for all rows, all columns, first row, last row etc.

This definition is not very exact, therefore we will explain it in more detail: When we have a specific shape in our building, the spatial locator should define an unambiguous position of this shape in order to separate it from other instances of this shape. Therefore the locator

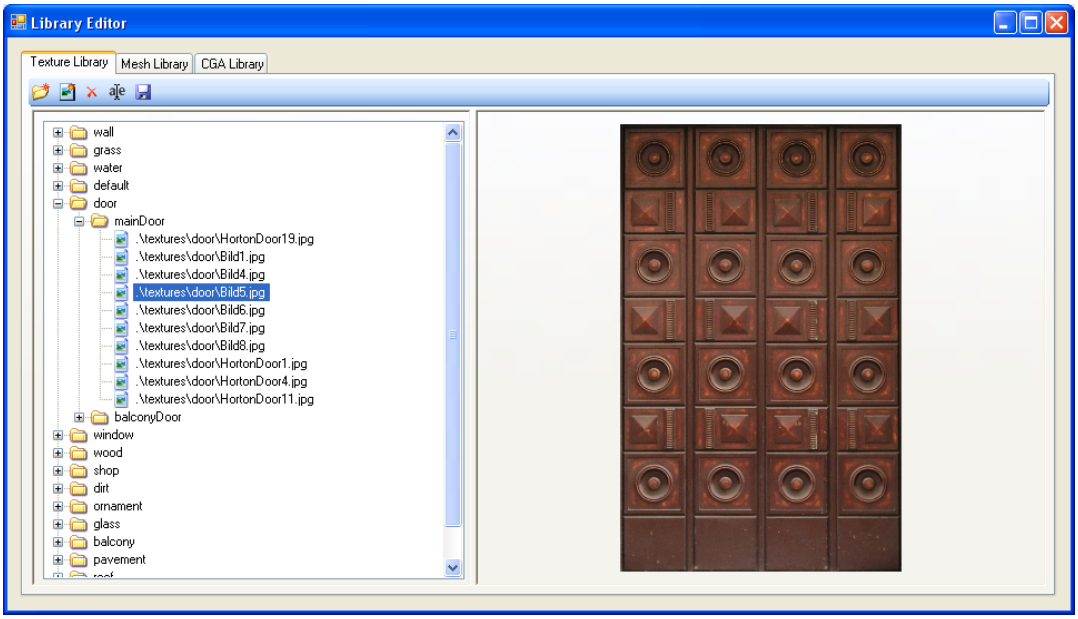

Figure 4.8.: Texture Library. On the left hand side, the hierarchy of texture categories is displayed. Every folder icon represents a category. The user can create, rename and delete categories using the buttons above. Textures can be moved from one category into another with drag-and-drop operations. Every texture belongs to one category.

should include facade number, floor number and column number. When we have nested floors or rows, a subfloor and a subcolumn has to be specified too.

We use this locator to identify positions of parameter changes and dropped elements. During the production application process, the locator values (façades, floor, row) must be determined for every shape. This is done the following way:

When a repeat command is used to create multiple floors, the program creates the subtree parameter `floorNumber` for every floor and automatically assigns values. However, how should the program determine which repeat command actually creates different floors? We have to help the program by explicitly specifying when a split or repeat command creates multiple floors. This is done the following way:

```
facadeB ~  Repeat(Y,1;floorNumber){floor};
```

Here, `floorNumber` explicitly tells the program that this repeat command creates multiple floors. Now, the program can automatically assign the subtree parameter `floorNumber` to

every floor. Of course, we also have to specify this for façades and rows. Here is an example specifying a spatial locator:

```
house ~ Comp(FACES,SIDE;facadeNumber){facade};
facade ~ Repeat(Y,1;floorNumber){floor};
floor ~ Repeat(X, 1;columnNumber){tile};
tile ~ ... ;
```

With this code, every instance of the production with predecessor `tile` has the parameters `facadeNumber`, `floorNumber` and `columnNumber` unambiguously defined.

At the moment, only the parameter names `facadeNumber`, `floorNumber`, `columnNumber`, `subFloorNumber` and `subColumnNumber` are allowed. Parameters with other names are ignored. To increase flexibility, this restriction will be removed in further versions.

It is not necessary to specify every parameter, for example `subFloorNumber` can be omitted as seen in the previous example. When a parameter is omitted, it is initialized to `-1000`.

Here is an example spatial locator, serialized into an XML element:

```
<selectionDefinition floor="1" subFloor="-1000"
   column="2" subColumn="-1000" facade="1" />
```

The XML elements associated with spatial locators are called `selectionDefinition`. The suffix `Number` is omitted for parameter names.

Persistence of User Decisions When directly modifying text files, the persistence of buildings poses no problem: As the grammar is modified, the same building can be generated at a later time with this grammar. However, when we apply changes via the GUI, we need to determine where to save those changes, in order to load them later.

One possibility would be to automatically change the underlying CGA code to reflect the changes. This would compare to an automated version of the text editing method. However,

we want to go one step further: We want to allow user decisions without modifying the underlying CGA code at all. This is advantageous when multiple buildings are generated with the same grammar. This way, instead of creating a copy of the grammar for each building and modifying this copy, we can leave the grammar as it is and save the building specific changes elsewhere.

We solve the problem where to actually save the changes in the following way: All user decisions are saved with their corresponding spatial locator. This information can be serialized into an XML file. Now, during production application we search this data for a match with the current location. When such a match is found, the saved user decision is used instead of the production rule. This way user decisions are retained.

Persistence with Multi Level Editing One advantage of CGA is the possibility to make design decisions on multiple levels [WWSR03] [MWH$^+$06]. For example, coarse decisions, like the age of the building, can be set at the beginning of production derivation. At later stages, specific parameters for windows can be set.

Suppose the user applies coarse decisions on the whole building and some fine decisions on the window. When coarse decisions are changed later, a persistence problem occurs: We do not want to lose the fine decisions applied to the window. However, the window may get changed implicitly after the coarse decision:

1. The window may not exist anymore, for example when the building size was reduced.
2. The window may get changed to a different part, for example when a floor type was changed.

When fine-grained user decisions exist for the window, they should be retained. In the first case, the user decisions must not be deleted. It should be saved for later retrieval, in case the window is used again later.

The second case is more difficult: As many user decisions as possible should be transferred from the window to the new part. For example, if the user set a specific size for the window, the new part should retain this size.

The first possibility is easily handled both using textual editing and GUI modifications: Instead of deleting the old part, it is saved externally.

However, the second possibility can not be easily solved with textual editing: In order to retain decisions, the new production rule needs to be rewritten to reflect the fine-grained user decisions.

With GUI editing, and external storage of user decisions as described in the previous paragraph, this case can be handled rather simply: During production application, an automatic search of the user decisions for common parameters is conducted. When parameters match, they are used for the new part. This way, fine-grained user decisions done for the window are retained as much as possible.

4.2. Language Enhancements

In this section, additions and improvements of the CGA syntax are proposed. At first, we will introduce a new concept called *subtree parameters*. Afterwards, we will focus on two subjects that were partially solved in [MWH+06]: *Production variations* and *volumetric shapes*. We feel that more exact definitions of those two subjects are necessary.

4.2.1. Subtree Parameters

To parametrize shapes in CGA, global parameters or parameters local to a production are possible [MWH+06]. They were described in Section 3.3.1. A global parameter naturally has a scope over all productions. The scope of a local production parameter is only valid for this production and does not extend to successors of this production. For example, consider the following productions:

```
A ~ B window(10);
window(localVar) ~ C D;
```

`localVar` is a local production parameter of window. It is visible during application of production with predecessor `window`. However, when `C` or `D` are applied, `localVar` has lost its scope. In Figure 4.9 a possible derivation result is shown, depicting to which productions `localVar` is visible.

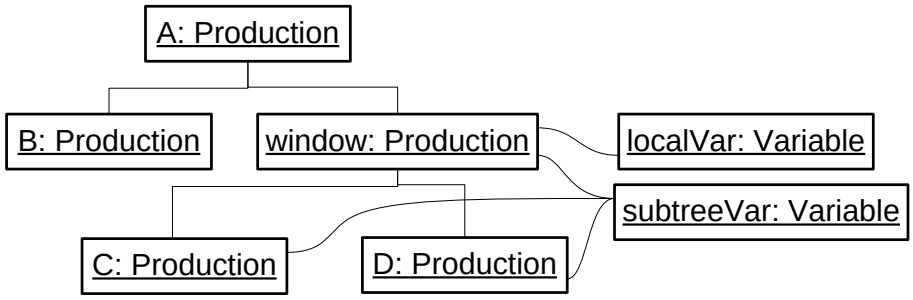

Figure 4.9.: UML object diagram showing possible derivation

This locality of the scope allows for a fine-grained control over shape parameters. However, as the scope does not extend to successors, there is a problem when trying to implement the following case: Consider the user wants to create a parameter for the window that is also used for some successors of the window. In order to get the parameter to all window successors, the following code would have to be employed:

```
A ~ B window(10);
window(localVar) ~ C(localVar) D(localVar);
```

Clearly, this is not a good idea, as all successor productions need to be modified to take along the parameter. Another possibility is to use a global parameter instead of the local one: Now all successor elements see the parameter. However, the fine-grained control is lost using global parameters, as all windows need to use the same parameter.

Therefore, a method that allows fine-grained control and provides parameter access to successor elements is proposed, named *Subtree Parameters*. These parameters extend their scope to all elements that are subsequently derived from this production. The following code shows how these parameters are employed in CGA:

```
A ~ B AssignVariable(subtreeVar,10) window;
window ~ C D;
```

Figure 4.9 shows which productions can see the parameter. Using this code, the previously mentioned problem is solved: All successor elements see the parameter, while it can still be defined for each window separately. The user can also override a subtree parameter: Issuing another `AssignVariable` command overwrites the value for the corresponding subtree.

4.2.2. Production Hierarchies

As described in Section 4.1.2, we need hierarchies to determine applicable elements for drag-and-drop operations. For textures and meshes, those hierarchies can be easily defined by the user, as seen in Figure 4.8. For production hierarchies this is not that simple. There are two possible ways a hierarchy of productions could be established.

On the one hand, the program could try to analyze the productions and automatically search for implicit hierarchies. On the other hand, the user could explicitly specify hierarchies of productions. Therefore we would need to extend the CGA syntax.

Let us have a look at how implicit hierarchy search could work: Multiple choices for one production name are possible in CGA, by using the same predecessor name, and applying probabilities to each production [MWH+06]. For example, multiple window types are defined with the following CGA code:

```
window ~ typeA: 0.5;
window ~ typeB: 0.5;
```

Using this syntax combined with intermediate production rules enables hierarchical production variations:

```
window ~ typeA;
window ~ typeB;
typeB ~ typeC;
typeB ~ typeD;
//actual production definition
```

```
typeA ~ ... ;
typeC ~ ... ;
typeD ~ ... ;
```

As shown in Figure 4.10, this could be interpreted as an implicit definition of a production hierarchy. However, to determine those hierarchies, the program would have to perform some sort of graph search of production dependencies. We do not have this functionality implemented, and thus can not provide an exact algorithm that is able to do this.

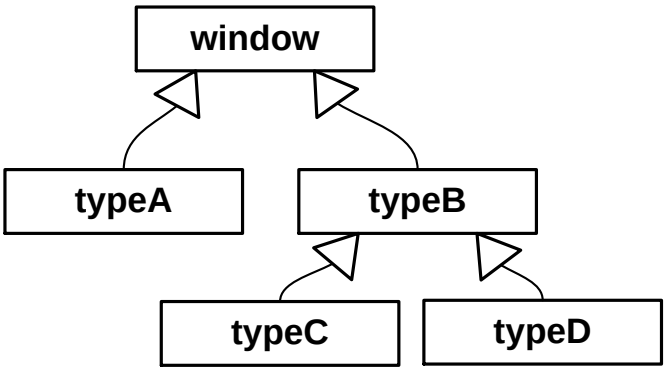

Figure 4.10.: UML class diagram of implicit production hierarchy

Explicit hierarchy definitions do not require this search by the program. We propose the following syntax to define explicit hierarchies:

```
childType{parentType} ~ ... ;
```

Our previous example can thus be written as:

```
typeA{window} ~ ... ;
typeB{window} ~ Epsilon;
typeC{typeB} ~ ... ;
typeD{typeB} ~ ... ;
```

Note that the syntax `typeB{window} ~ Epsilon;` explicitly defines typeB as an abstract type that can not be used as terminal symbol in an derivation. Using this syntax, an exact hierarchy is specified. There is no need for the program to perform searches, as the parent of each production is known. This makes hierarchy generation trivial.

To sum this section up, implicit hierarchies have the same expressivity as explicit hierarchies. There is no extra functionality of an explicit hierarchy. However, implicit hierarchies would require a special graph search that may be non-trivial to implement. With our new syntax for external hierarchies it is trivial to build the hierarchy tree, as the father of each production is known. Further, the user has total control of the generated hierarchy. Therefore we think explicit hierarchies are a better choice.

4.2.3. Volumetric shapes

The idea to use combinations of volumetric shapes is introduced by Müller et al [MWH+06]:

> Modeling strategy: ... First, we use three-dimensional scopes to place three-dimensional shapes (volumes) to form a mass model ...

The volumetric shapes are then used for subsequent component splits. However, how to actually define the placement of these nonterminal shapes using CGA syntax is not specified by Müller et al. in their publication [MWH+06]. Only the placement of terminal shapes with the `I()` command was specified.

To eliminate the uncertainty of this issue, a new CGA language element `VolumeShape (CYLINDER | CUBE, detailLevel)` is proposed. When this element occurs, a volumetric shape is explicitly created and used for subsequent component splits. The following code shows an example on how this new CGA construct can be used to define multiple cylindric shapes, Figure 4.11 shows a house generated using this code part:

```
petronas{houseType} ~ S(1,houseHeight  a,1)
   Comp(FACES,SIDE;facadeNumber){facade}
   Comp(EDGES,BOTTOM;facadeNumber){petronaCylinder};
```

```
petronaCylinder ~ S(10a,houseHeight  a,10a)
   VolumeShape(CYLINDER,6)
   Comp(EDGES, TOP){fence}
   Comp(FACES,SIDE;facadeNumber){facade};
```

Please note that the new language construct does *not* add new functionality, it only clarifies how to define multiple volumetric shapes.

Figure 4.11.: Example using VolumeShape

4.3. Summary

We have shown how our interactive editor can increase the usability of CGA. Our editor helps by providing drag-and-drop operations for production variations. Further, all parameters are automatically mapped to sliders, making modifications simple. It is possible to change parameters for subsets of the building. The ground-plan vertices are movable, and the building rendering is updated instantly. Also, all textures, meshes and productions are sorted into libraries for better reusability.

Afterwards, language enhancements were described. These include subtree parameters, an orthogonal definition of production variations, and specific volumetric shape commands. Those enhancements help in creating a more robust CGA implementation.

Chapter 5.

Implementation

While the former sections provided an overview on theoretical aspects of shape grammars, this section will explain implementation details for computerized shape grammars.

At first, the important aspects of the software architecture are described. The Unified Modeling Language (UML) [OMG07] [MH06] is employed for this task. Afterwards, performance enhancing techniques used in the implementation are explained.

5.1. Software Architecture

In this section, the software architecture will be described. At first, a rough overview over the system is provided. Further subsections go into more detail on specific parts of the system. Note that it is neither possible nor sensible to describe all aspects of the system in this thesis, only the most important parts are explained here. For a full description, the documentation of the source code needs to be used.

5.1.1. System overview

In order to keep the system flexible to further changes, functionality is distributed into different modules. Figure 5.1 displays those modules.

The module `CGA` contains all CGA related functions: The `Parser` reads `.CGA` files into `Nodes`. `CgaManager` combines parsing results of multiple files into one hierarchy of `Nodes`, and performs semantic analysis over the combined parsing results. During the semantic analysis, symbols are resolved. Section 5.1.2 focuses on parsing and semantic analysis in more depth.

`CgaMain` further provides methods to apply the productions to a given axiom. `Visitors` implement the functionality of production application, command application and rendering of the production result. The process of production application is explained in Section 5.1.3

`Rendering` provides methods to initialize OpenGL, manage textures and meshes, and draw the scene with special effects.

`GUI` contains `GuiAppFacade`: This is the main class for handling communication between the GUI and CGA. It implements the façade design pattern [GHJV95]: A unified interface for CGA and Rendering functionality is provided to be used by all GUI components. `Form1`, the main GUI form, uses the interface of `GuiAppFacade`, and never directly accesses other modules. This way, the GUI is separated from CGA and Rendering.

5.1.2. Parsing and Semantic Analysis

Müller et al. provided an informal description of the CGA language [MWH+06]. In order to write a stable parser, a more formal definition is required. This definition is provided in Appendix A: Both the syntax described by Müller et al. [MWH+06] and new language constructs are represented in Backus-Naur form.

The actual parsing process is performed using the `Boost::Spirit` parsing library [boo07]. With this `C++` library, a direct specification of the language syntax, using an EBNF-like notation, is possible. During parsing, an object from a class hierarchy representing the parsed data is instantiated for each CGA statement. Figure 5.2 depicts the implemented class hierarchy.

The usage of those nodes is as follows: `Production` nodes represent a CGA production, `Rule` nodes describe CGA commands and `Shape` nodes denote CGA commands that result in renderable objects. Please note that the name `Rule` is confusing, as it conflicts the

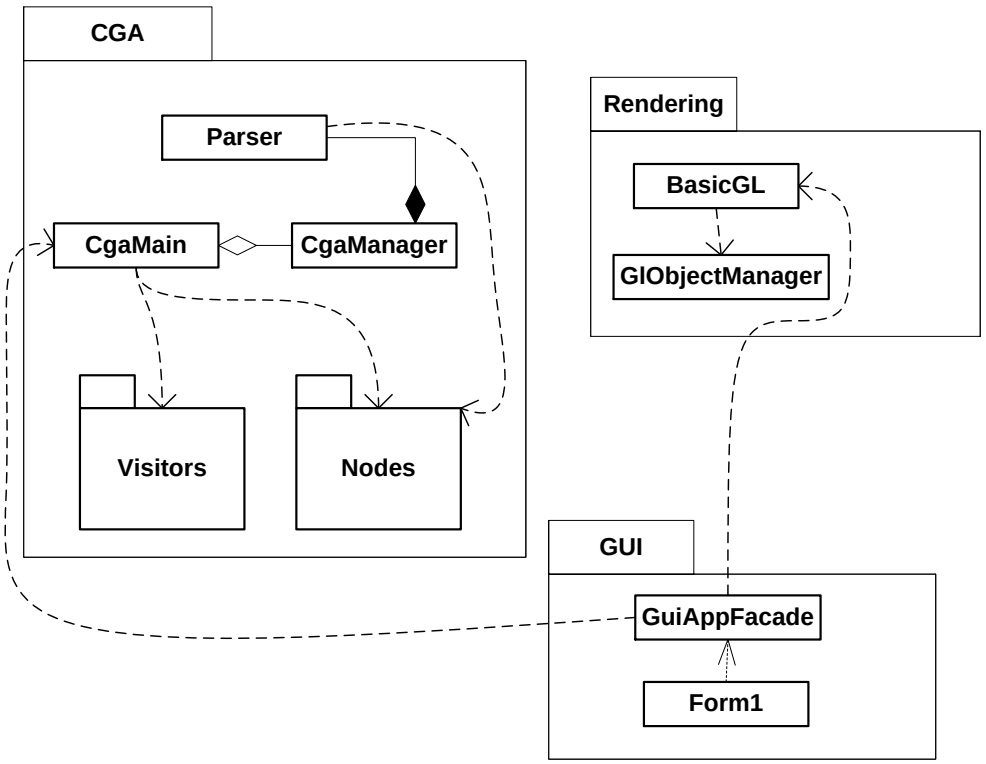

Figure 5.1.: UML class/package diagram of main system

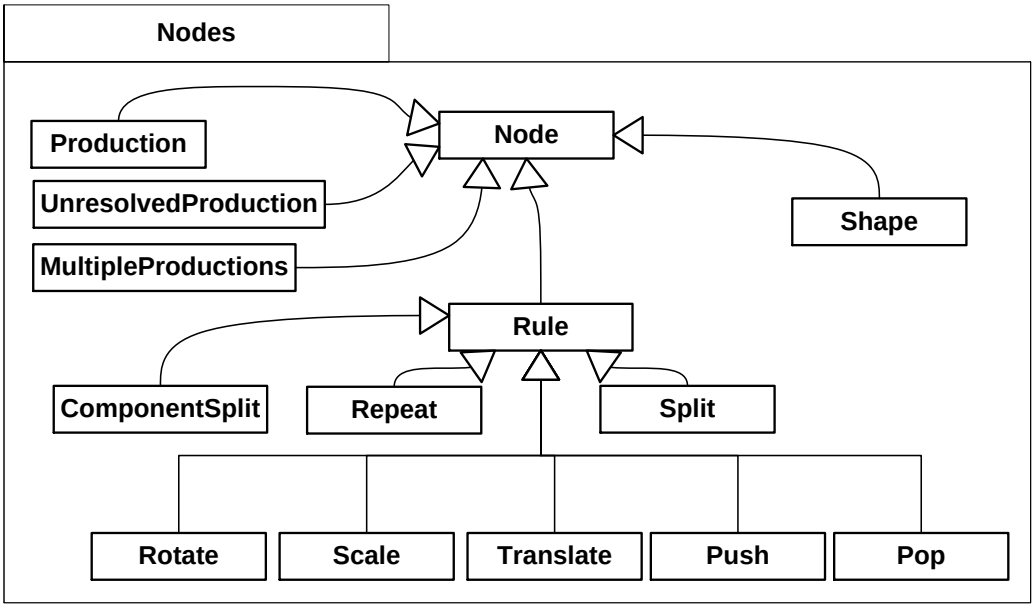

Figure 5.2.: UML class/package diagram of Nodes

typology used in this thesis, and will therefore be replaced with `Command` in further versions. `Shape` also conflicts with the defined typology, `TerminalShapeCommand` would be more appropriate. Those two inconsistencies where caused by a misunderstanding during an early development cycle.

There are different types of Production nodes implemented. The rationale behind this is explained here. `Production` nodes store the following data: Successor nodes, priority and probability. In contrast, `UnresolvedProduction` nodes occur in the successor of a `Production` node. They just contain the name of the production and a parameter list. If the production is applied, the `UnresolvedProduction` nodes have to be resolved first.

We could handle `UnresolvedProduction` as non-terminal nodes, and resolve them every time they are encountered during derivation. However, in order to speed up the derivation, we choose to resolve all possibilities before starting the derivation.

Therefore a `MultipleProductions` node is created. This node contains references to one or more `Production` nodes, as a production name may refer to multiple productions. To make this usage pattern more apparent, refer to Figure 5.3 for an example object config-

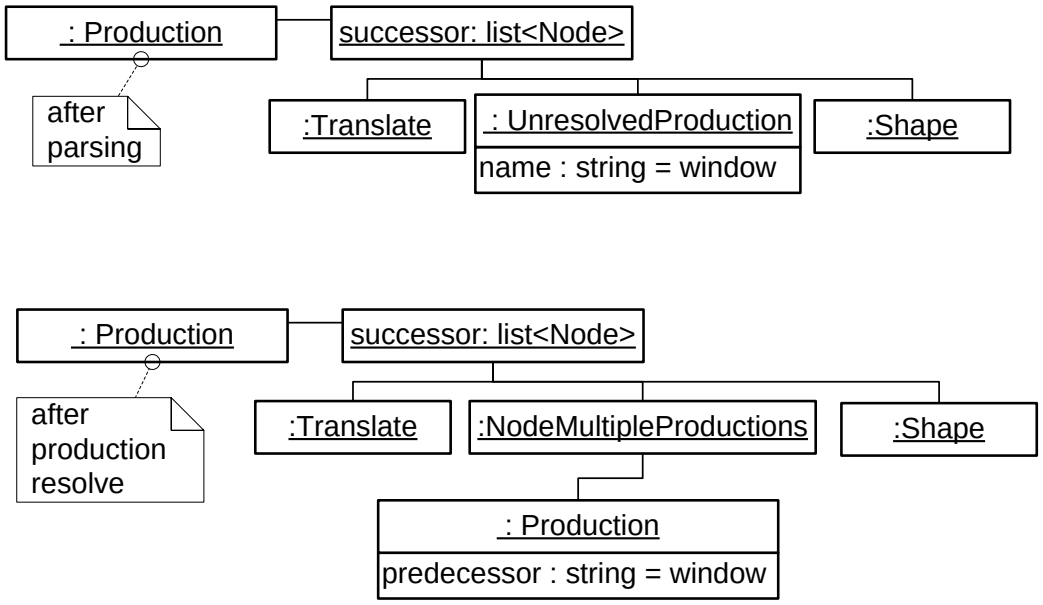

Figure 5.3.: UML object diagram of Production node usage

uration: The first diagram represents a possible parsing outcome, the second diagram shows how `UnresolvedProduction` nodes are replaced with `MultipleProductions` nodes.

After all CGA files are parsed, semantic analysis has to be performed to ensure consistency of productions. Semantic analysis consists of the following steps:

1. The productions, parameters and shapes of all CGA files are collected into one common data structure.
2. Duplicate entries in this data structure are eliminated.
3. The hierarchy of productions is generated.
4. Parameters representing numeric values are resolved. When the parameter is not defined, the corresponding production is removed.

Note that the replacement of `UnresolvedProduction` with `MultipleProductions` nodes is not performed during semantic analysis. It is done later, when the actual productions are defined.

5.1.3. Production Application

The visitor design pattern [GHJV95] is used to encapsulate the different functions that need to be performed on nodes. Figure 5.4 shows an UML diagram of the implemented visitors. `ProductionVisitor` performs the application of productions. A priority queue is used to determine the order of production applications.

`RuleVisitor` is called by `ProductionVisitor`. Its task is to perform all operations described in Nodes derived from `Rule`.

`RenderVisitor` has three different passes: Collect nodes, calculate transformations and render. The first pass is called during production application. All shape nodes are collected into lists sorted by their texture. The second pass is performed afterwards: According to the node scope, the transformation matrices used for rendering are calculated. For each rendered frame, the last pass is called: At first, the corresponding texture is bound, then shape geometry is rendered.

Note that the visitor design pattern allows encapsulating traversal methods over the node tree. This would allow one traversal method to be used by all visitors. However, this encapsulation possibility is not used, as each visitor needs a different kind of traversal. Therefore each visitor implements its own node traversal algorithm. `ProductionVisitor` traverses nodes using a priority queue, `RuleVisitor` recursively traverses all command nodes in a specific production, and `RenderVisitor` linearly iterates over all collected shape nodes.

5.2. Performance Optimization

Previous publications did not focus on performance related details. This is probably caused by the lack of an interactive design paradigm: A compile cycle was needed for each change of a parameter. Therefore direct visual feedback in real time is not necessary.

However, for an *interactive* building editor, direct visual feedback is important, and therefore performance does matter.

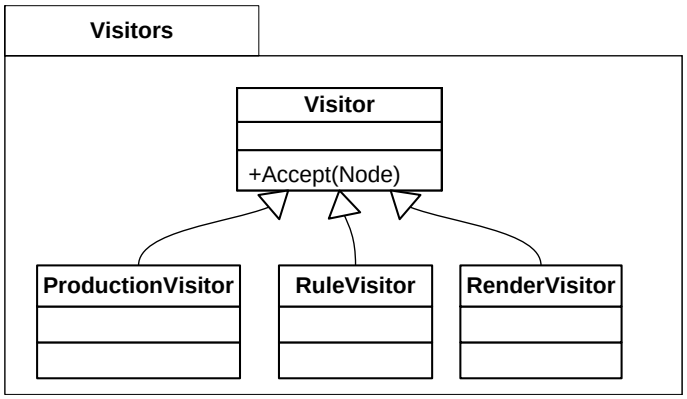

Figure 5.4.: UML class/package diagram of Visitors

At first, hardware-accelerated occlusion queries are introduced. Then, object pools are explained. Afterwards, an application of the flyweight design pattern is proposed. In Section 5.2.4, rendering accelerations using OpenGL pseudo instancing are explained. Finally, possibilities to optimize some operations are pointed out.

5.2.1. Hardware-Accelerated Occlusion Queries

In order to get fast occlusion tests between shapes, the hardware-accelerated algorithm proposed by Knott [Kno03] is employed. It is a combination of stencil buffering and hardware occlusion queries. Basically, the building is interpreted as an shadow volume, and the z-Pass algorithm is used to search for shapes intersecting this volume. Our slightly modified version works as follows:

1. During production application, all productions containing an occlusion test are saved in a list for later resolving.
2. When no productions with a higher priority than the occlusion test priorities are left in the queue, a batch of occlusion tests is performed.
3. The productions corresponding to the test result are added into the priority queue.

The actual occlusion queries are described next. Note that windows are used as an example:

1. All elements are sorted according to their normal vector. Then all elements having the same vector are sorted in coplanar groups. This way, we can reduce the amount of rendering passes.

2. Now, for every group, an orthogonal view port showing all nodes is created.

3. Using this view port, all windows are drawn to establish the z-Buffer.

4. Now the building is drawn, using the following stencil configuration: Increase on back faces, decrease on front faces. This way, for every pixel of an unoccluded window, the stencil value should be zero.

5. To actually test if a window is occluded, we draw each window again, testing $stencil! = 0$, and issuing an occlusion query for every window. If the query result is higher than zero, at least one pixel of the window was occluded.

We can not provide performance comparisons of this algorithm with the octree-based method used in [MWH+06], as we only implemented the hardware-accelerated variant. Results of of our variant can be seen in Section 6.2.

5.2.2. Object Pools

During the application of productions, a huge amount of objects are created and deleted, taking a considerable amount of time, as seen in Section 6.2.

To alleviate this performance problem, object pools are used. The main idea is to batch `new` calls. Figure 5.5 shows how they are implemented: A template class `ObjectPool` manages many objects of type `T`. When `GetNewObject()` is called, an object from `objectPools` is returned. When no new elements are left, `ObjectPool` automatically creates `batchSize` new elements. The class `ProductionResult` stores the derivation tree. For every object type an object pool is employed.

The results of this optimization can be seen in Section 6.2.

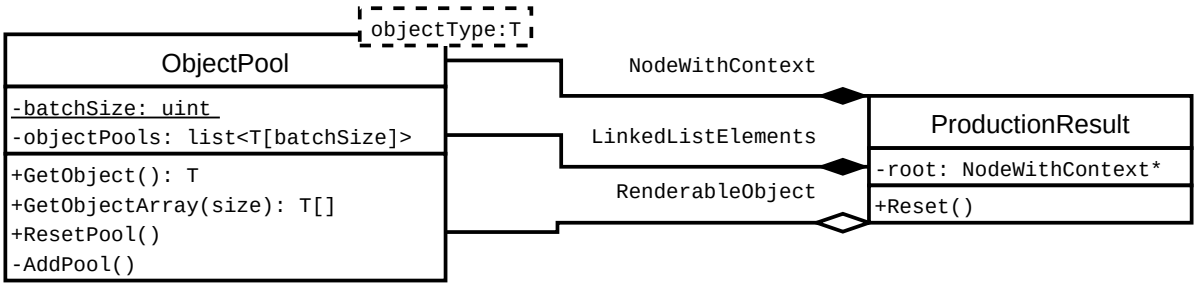

Figure 5.5.: UML class diagram of object pools

5.2.3. Flyweight Design Pattern

As the application of productions proceeds, one node in this structure may be represented many times in the derivation, every time with different parameters and scope. For example, a wall tile may occur all across the building. A naive approach would be to create a copy of this tile for every occurrence with the local parameters. This would incur two problems: At first, the memory requirements are increased, as some aspects of the tile are stored in multiple positions. Another problem is consistency: Multiple storage of equal values may lead to inconsistent results, and global changes are not easily possible anymore.

To alleviate these two problems, we apply the flyweight design pattern. Therefore we divide node parameters in extrinsic and intrinsic state. Intrinsic state includes properties that do not change over multiple instances. For example, this may be the definition of the repeat-with for the repeat command. Extrinsic parameters are different for multiple instances. This includes the scope for all nodes. Further, the derivation information, like father and child nodes, are extrinsic state for all nodes.

All this extrinsic state is saved in the class `NodeWithContext`. During derivation, only objects of this type are generated, nodes are never directly generated. Figure 5.6 shows a UML diagram representing the described structure.

As this pattern was incorporated at the very beginning, we can not provide performance comparisons to an approach without this pattern. However, for a typical building with N windows, we estimate memory savings of approximately N times, as only one instance of the windows has to be stored.

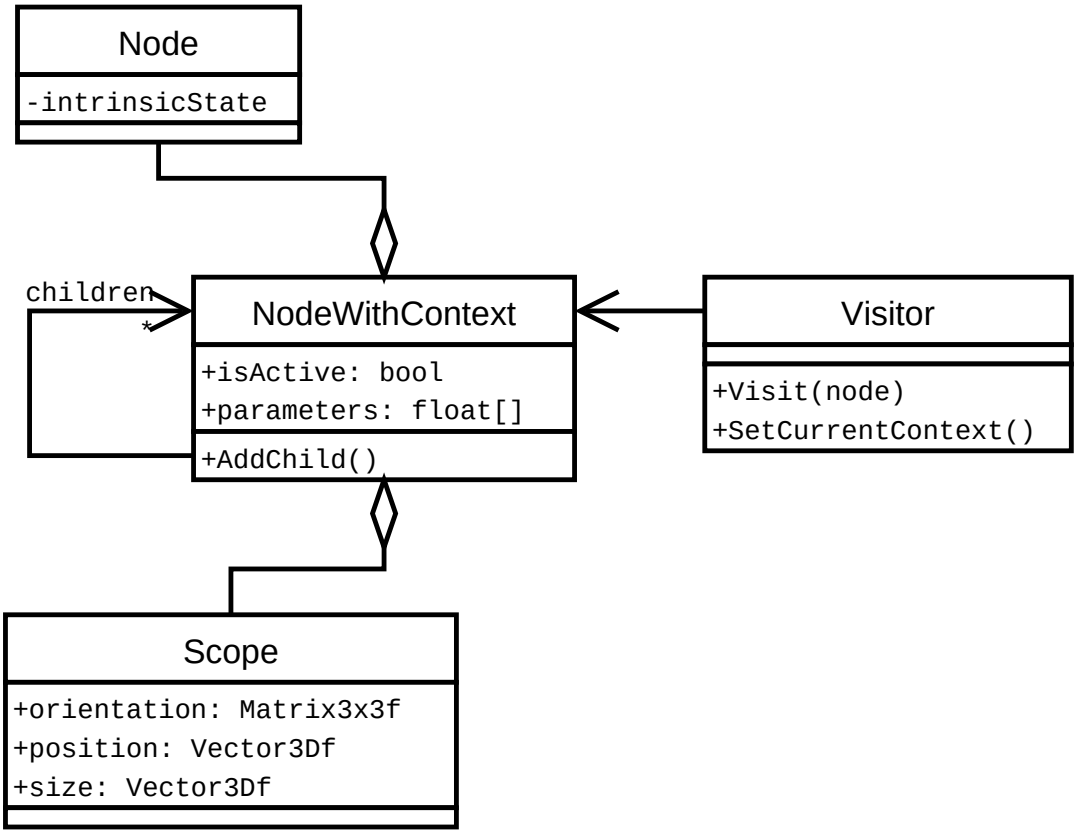

Figure 5.6.: UML class diagram of implemented flyweight pattern

5.2.4. OpenGL pseudo-Instancing

The derivation tree data structures are optimized for flexibility: They are very fast at new and delete operations. Thus, generation of buildings is fast. However, this is in contrast to requirements for fast rendering: Fast rendering would require to create as few static batches as possible for the building, and loading them into vertex buffers. This would incur high costs to update the vertex buffer each time the building is generated.

Therefore we make a compromise: We directly use the many small parts generated during CGA application utilizing instancing. OpenGL pseudo instancing [Zel04], a method to speed up instanced geometry rendering, allows us to draw small parts many times with different texture and model view matrices.

The use of OpenGL pseudo instancing, together with sorting by texture, increased the rendering performance of the building in Figure 6.4 from 14fps to 42fps.

5.2.5. Optimized Sorting

After parsing the grammar, the amount of productions and possible textures is fixed. We can use this domain knowledge to optimize the following operations:

- During production application we can use bucket sort to create the priority queue.
- During rendering we can use bucket sort to sort our terminal objects by texture.
- All local parameters are mapped to a unique integer. This increases the performance of string comparisons needed to find applicable parameters, as we only need to compare two integers.

The impact of bucket sorting is evaluated in Section 6.2.

5.3. Summary

We have presented a flexible software architecture for CGA implementations. This architecture makes the addition of new commands easy by using the visitor design patterns. Then, details on the semantic analysis were provided. Finally, five techniques to increase the performance were proposed. The impact of those techniques is explained in the next chapter.

Chapter 6.

Evaluation

In this chapter an evaluation of the implementation is given. At first, the overall system performance is measured. Then the impacts of the optimizations from Section 5.2 are quantified.

6.1. Overall Performance

As our goal is an interactive editor, real-time performance is aspired. To evaluate if this goal is reached, we conduct performance tests. One aspects of the tests is to determine how the program scales with different building complexities. The following computer configuration was used for the tests:

Operating System	Windows XP SP2
CPU	Athlon XP 2600
RAM	1024 MB
GPU	Geforce 6600 128MB

Note that we have deliberately chosen a system that is a few generations behind the current hardware, to test if the performance is adequate for wide-spread configurations.

Three different benchmarks were conducted on different building sizes:

- **Building Creation:** A building is regenerated from scratch, but not rendered. This includes resetting to the axiom and applying all productions until the production queue

Triangles	Creation	Rendering	Creation + Rendering
32466	0.49	0.26	0.76
98002	1.55	0.74	2.31
150178	2.34	1.2	3.45
288770	4.46	2.38	6.76
375186	5.76	3.11	8.81
474346	7.24	3.9	11.05

Table 6.1.: Milliseconds per iteration versus triangles for various modes

Triangles	Creation	Rendering	Creation + Rendering
32466	203.95	382.16	131.34
98002	64.65	135.68	43.22
150178	42.82	83.59	29.01
288770	22.41	42.01	14.78
375186	17.37	32.18	11.35
474346	13.81	25.66	9.05

Table 6.2.: Iterations per second versus triangles for various modes

is empty. This has no direct correspondence to a user task, as generally the building is rendered too. However, this is still an important performance figure that allows optimizing the production applications.

- **Building Rendering:** An already generated building is rendered. This corresponds to a user changing the view on a building.
- **Building Creation and Rendering:** At first the building is generated, then it is rendered. This corresponds to a user modifying some parameters of the building using sliders, or dragging ground plan vertices. It is the most important performance number for interactive editing. Note that the results do not equal the addition of creation and rendering times, as parallelism between CPU creation and GPU rendering could occur.

The building size is varied using a global parameter. Figure 6.1 shows the different buildings. All tests were looped 100 times, then an average execution time was calculated. The raw values are presented in Table 6.1, using those values the iterations per second are calculated and shown in Table 6.2. Figures 6.2 and 6.3 were generated from this data.

Figure 6.1.: Building sizes used for the test

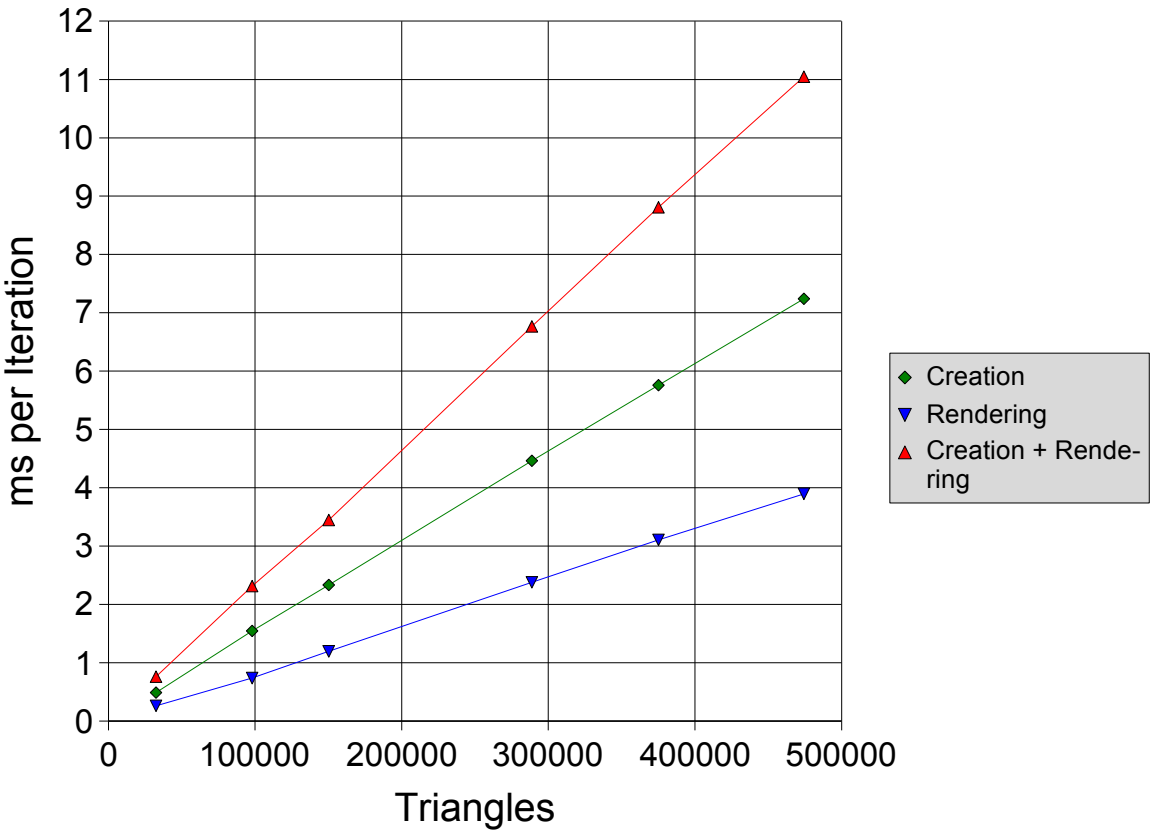

Figure 6.2.: Milliseconds per iteration versus triangles for various modes

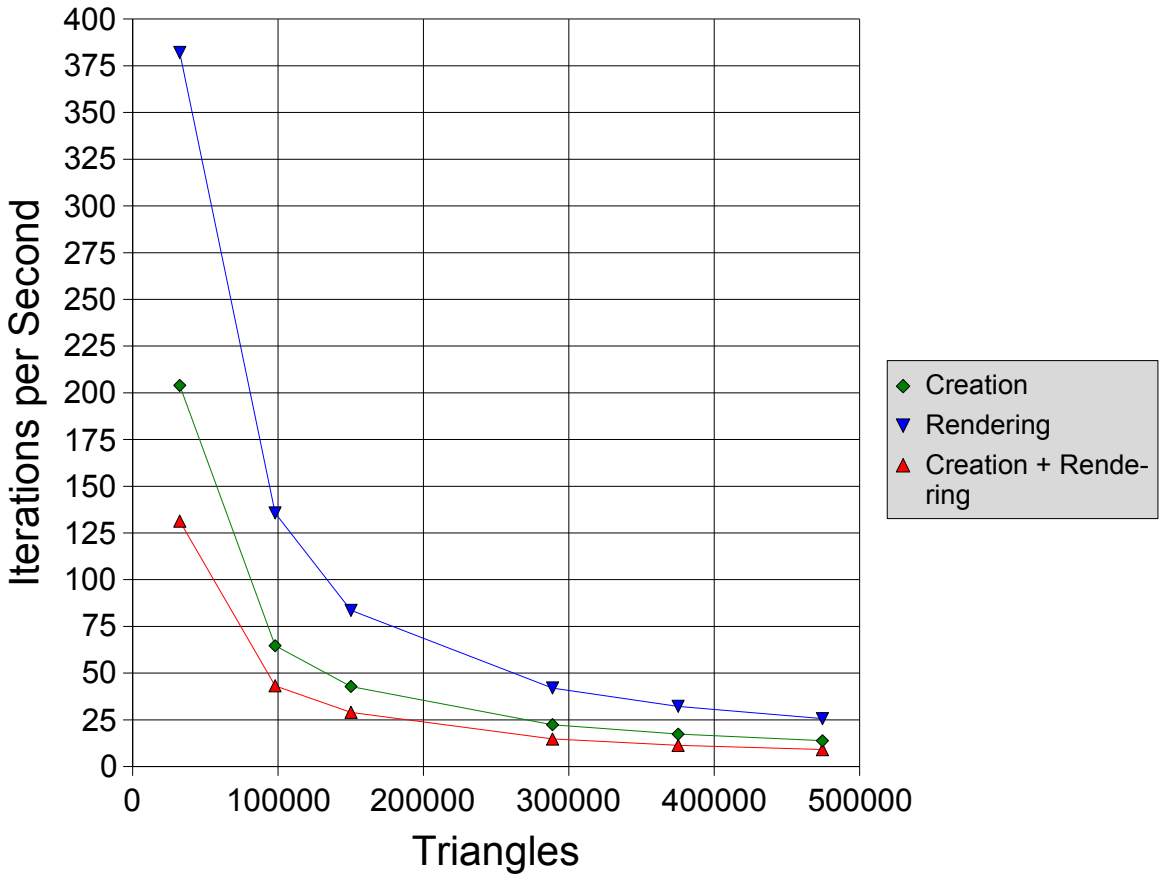

Figure 6.3.: Iterations per second versus triangles for various modes

Discussion We define real-time performance as 25 frames per second. In Figure 6.3 we can see that real-time performance for rendering is achieved for all tested buildings. However, the creation and rendering task is only real time for a building with about 200000 triangles. While those values are considerably higher than the ones reported in previous work (Müller et al. reported a few seconds to generate and display a building [MWH+06]), it is still advisable to search for additional optimization techniques.

Figure 6.2 shows that the performance scales linearly with the building complexity. This is beneficial, as a linear scaling provides a cushion for more complex buildings. Even better scaling could be achieved in future work by automatically reusing similar parts of a building.

As the addition of the creation and the rendering numbers roughly add up to the combined creation and rendering, we can deduce that there is no significant parallel execution of those tasks. Therefore additional performance can be gained by optimizing parallelism in future work.

6.2. Impact of Optimizations

The optimizations were implemented one after another. As the implementations are not orthogonal, and not easily switched on or off, it is not possible to provide detailed performance data for arbitrary combinations of optimizations.

However, during development, performance snapshots were obtained for different implementation stages. While we can not exactly quantify the impact of each optimization, these data still provide some insight. At first we will look on optimizations targeted at building generation, then hardware accelerated occlusion queries are analyzed. After each paragraph, the results are discussed.

Building Creation The building creation performance was evaluated for four stages:

- Initial implementation with the flyweight design pattern
- Addition of object pools, and replacement of std::list with custom linked list elements based on object pools

	ms per Iteration	new+delete	queue sorting	Random Nr.
Initial	8.61	79% (>50000 calls)	12%	1.92%
+Object Pools	1.29	0.9% (180 calls)	56.00%	11.70%
+Bucket Sort	0.62	1.70%	0.66%	26.98%
+Boost Random	0.38	0.40%	1.09%	0.42%

Table 6.3.: Performance values for different development stages, in percent of total execution time.

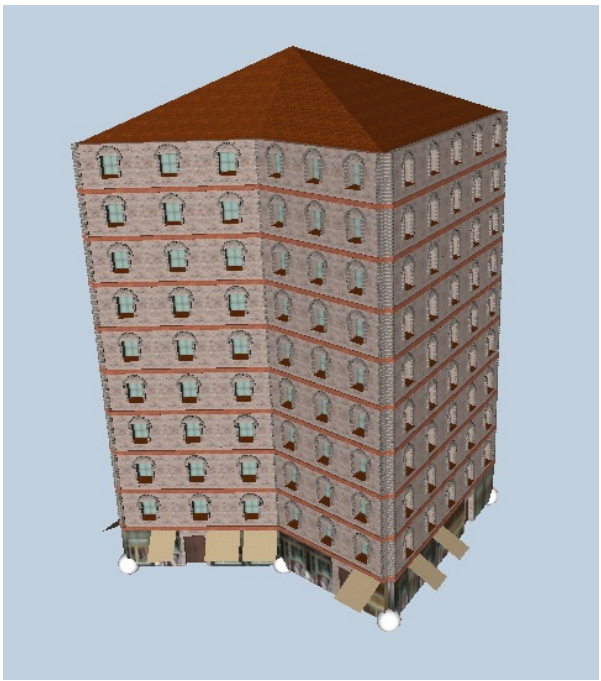

Figure 6.4.: Building used for performance profiling

- Bucket sort for priority queue sorting (discretization)
- Replacement of rand() with boost lagged fibonacci random generator [boo07]

After each stage, a detailed performance report was obtained with IBM Rational Purify [pur07]. This report was then used as a basis for further optimizations. Table 6.3 shows excerpts from the Purify report. Figure 6.4 shows the building used for profiling.

Building Creation Discussion The first thing to note is that object pools are a very important optimization. The amount of new and delete calls is reduced from 50000 to 180,

	Triangles	Creation	Rendering	Creation+Rendering
Queries	44356	29.1955	146.0618	24.28053
No Queries	86678	38.24582	83.67018	25.72194

Table 6.4.: Iterations per second versus queries/no queries.

corresponding to a 79% versus 0.9% execution time. The total performance improvement, including the replacement of std::list with custom lists, is 85%.

Using an efficient sorting algorithm (from bubble sort to bucket sort) for the priority queue adds another 52% of performance.

Finally, as seen in second last row, the standard random number generator requires a disproportional amount of performance. Therefore it is replaced with an optimized version, yielding another 51% of performance improvement.

In my opinion, the most important lesson to be learned from this section is that performance profiling is an essential part of the optimization process. For example, I did not expect that the standard rand() function takes so much time, but through profiling this was pointed out.

Hardware Accelerated Occlusion Queries The tests were performed on the building seen in Figure 3.11 and 4.11. Note that we can not provide comparisons with the unaccelerated octree version proposed by Müller et al. [MWH$^+$06], as only the accelerated version was implemented. Table 6.4 shows the obtained results.

Occlusion Queries Discussion One important thing to note is that the version with queries has significantly less triangles, as some windows are culled. Therefore the rendering performance is higher when queries are enabled. However, the creation itself takes longer when queries are enabled. The total results are about the same, therefore we can say that our occlusion approach is able to deliver real-time performance.

Note that the performance compared to the triangle count is lower for this building than for the building presented in Section 6.1. This is because the occlusion test building has less complex windows, and therefore a higher relative overhead during rendering occurs.

6.3. Summary

This chapter can be summarized into four key findings:

- For moderately sized buildings, real-time modifications of parameters are possible
- Rendering is possible in real time even for large buildings
- Building generation and rendering scales linearly with triangle count
- Object pools provide a major performance improvement

These findings imply that our main goal, real-time editing of buildings, is partially met. While the performance is very good compared to older implementations there is still need for improvement, in order to handle very large or more complex buildings in real time.

Chapter 7.

Conclusion and Outlook

At first, we will summarize the findings of this thesis. Afterwards, an outlook on possible future work is provided.

7.1. Conclusion

In this thesis, we provided an overview of procedural modeling techniques applicable to architecture. Four grammar-based approaches, shape grammar, split grammar, L-systems and CGA were explained. The adjacent research areas have been highlighted.

We found some disadvantages of CGA, and based the thesis focus on alleviating them. Do determine if we were successful in this task, let us restate the thesis objectives:

1. Create a complete CGA implementation from scratch, as we do not have a code base to work on.
2. Increase the usability of CGA
    - Allow interactive editing
    - Provide artist control
3. Show techniques to increase the performance of CGA to allow real-time manipulation of parameters

The first objective is reached. The only exception are snap lines, they will be included in future work.

We made some steps in the right direction in order to reach the second objective. It is important to note that usability is a very vast area, and it is impossible for one person to create a perfect solution, a team of software engineers would probably be required. However, there are some considerable improvements over the old approach: A unified GUI is provided that allows to visually modify buildings and parameters in real time, without resorting to text editing. However, new production rules still have to be defined in text form. In future work we want to resolve this.

As we have seen in the previous chapter, the third goal is reached too, at least for moderately sized buildings.

Independently of those goals, other areas of CGA were improved. Subtree parameters were introduced, we feel that they increase the flexibility of parametrization, as parameters can be defined on multiple hierarchy levels.

7.2. Outlook

Of course there are many things left to do if CGA has to be applied in commercial products. We will now point out some future work, that needs to be addressed, in order to reach this goal:

Towards Complete City Generation At the moment, single buildings are generated. In order to create whole cities, connections to the areas outlined in Section 1.7 need to be established. For example, the result of an allotment algorithm can be used as ground plans for multiple buildings. The user should have the possibility to modify the lots, and assign specific attributes and buildings to them. Further, changes in the allotments should retain as many user assignments of buildings as possible. Adding this functionality directly into the existing GUI would be an interesting research topic.

Towards Real-Time Rendering for Games While our framework enables real-time performance for individual buildings, as seen in Chapter 6, there are still some things to do if those buildings should be used in a real-time game. This includes:

- Faster building rendering. Now, one building is rendered in real time. However, in games, many buildings together should be rendered that fast. The first thing to try would be baking the building into fixed vertex buffers, using a GPU cache friendly vertex layout. This should increase the performance considerably.
- Automatic LOD generation: In order to render large scale scenes, low detail versions of the building have to be created automatically. Another option would be defining the low resolution versions directly in the grammar.
- Mesh cleanup: At the moment, the mesh may contain T-Vertices and coplanar polygons. In order for other rendering techniques to work, for example light mapping, those issues need to be resolved.
- Automatic generation of a collision detection acceleration structure, for example an oriented bounding box hierarchy. It should be easy for a game to use this hierarchy.
- Support for decals (e.g. bullet holes) everywhere on the building.
- Possibility to assign interactivity and game logic to a building. For example, doors should swing open when a player touches them.

Towards Full Graphical User Interface As described in Section 4.1, a few steps towards a unified graphical interface were already done. However, the grammar still has to be defined in text form. Therefore an interesting future work would be to create a graphical grammar editor. This grammar editor should enable artists who never used a scripting language before to use our tool.

Increasing the Expressivity of CGA At the moment the CGA commands split and repeat only look at the current scope to determine their result. The current shape configuration is not queried. An interesting future work would be to create commands that directly use the current shape configuration to determine their output. This would for example make splits of

triangle-shaped building shells into multiple floors possible, and thus increase the expressivity of CGA.

Appendix A.

Formal language definition

The Backus-Naur form of the accepted language will be shown here. To increase readability, quotation marks around terminal symbols are omitted for all terminals that are not equivalent to Backus-Naur constructs. For example, on the one hand, `Axiom` is no Backus-Naur construct, therefore the quotation marks are omitted. On the other hand, `"{"` is also used in Backus-Naur, and quotation marks are used to resolve ambiguities.

```
<program> ::=
   { <global_variable> | <priority> | <comment> }
   <axiom> { <production> | <priority> | <comment> } ";" ;

<axiom> ::= Axiom ":" <node> ";" ;

<global_variable> ::= <symbol> "=" <real> ";" ;

<priority> ::= Priority "=" <uint> ";" ;

<comment> ::= ? C/C++ Style comments ? ;

<production> ::= <predecessor>
   [ "(" <symbol> { "," <symbol> } ")" ]
   [ ":" [<occlusion>] <condition> { "," <condition> } ]
   "~"
```

```
   <successor>
   [ ":"   <real>]
   ";" ;

<predecessor>  ::= <symbol>;

<contition> ::= <cond_binary>;

<cond_binary> ::= <floatfield> <binary_op> <floatfield>;

<binary_op>::= "<" | ">" | "<=" | ">=" | "==" | "!=";

<successor> ::= {<node>};

<node> ::= <scope> | <subdiv> | <repeat> | <comp>
   <insertobject> | <volumeshape> | <subtree> |
   ( <symbol> [<parameterlist>] ) | Epsilon;

<parameterlist> ::= "(" <floatfield> [{ "," <floatfield> }] ")";

<scope> ::= <translate> | <rotate> | <scale> | "[" | "]";

<translate> ::= T "(" <triple_floatfield> ")";

<rotate> ::= ( RX "(" <real> ")" ) |
   ( RY "(" <real> ")" ) |
   ( RZ "(" <real> ")" ) |
   ( R "(" <triple_floatfield> "," <floatfield> ")" );

<scale> ::= S "(" <triple_floatfield_abs> ")";

<comp> ::= Comp "(" <objType> "," <objPos> ","
   [<uint> { "," <uint> }] ")"
   "{" <node>  "}" ;
```

```
<occlusion> ::= ( Shape | Scope ) ".occ"
   "(" ( noparent | all ) ")"
   "==" ( none | part | full )

<objType> ::= FACES | EDGES | VERTICES ;
<objPos> ::= ALL | TOP | BOTTOM | SIDE ;

<subdiv> ::= Subdiv "(" <axes> ";"
   <subdiv_size> [{ "," <subdiv_size> }]  ")"
   "{" node [{ "|" <node> }] "}";

<subdiv_size> ::=  <floatfield> ["r"];

<repeat> ::= Repeat "(" <axes> "," <floatfield>
   [ "," ("t"|"f")] ")" "{" <node> "}";

<insertobject> ::= "I" "("
   ( CUBE | PLANE | CYLINDER | SPHERE )
   "," '"' <filename> '"' "," (t|f) "," <real> ă")";

<volumeshape> ::= VolumeShape "("
     ( CUBE | CYLINDER ) "," <floatfield> ")" ;

<subtree> ::= AssignVariable "(" <symbol> "," <floatfield> ")" ;

<axes> ::= [X] [Y] [Z];

<triple_floatfield> ::= <floatfield> "," <floatfield>
   "," <floatfield>;

<triple_floatAbs> ::= <floatfield> [a] ","
   <floatfield> [a] "," <floatfield> [a];
```

```
<floatfield> ::= <real> | <rand> | <symbol> |
   ( "(" (<real>|<symbol>) <arithmetic_op>
   (<real>|<symbol>) ")" );

<rand> ::= Rand "(" <real> "," <real> ")";

<arithmetic_op> ::= "+" | "-" | "/" | "*" ;

<symbol> ::= { [a-z] | [A-Z] } - <keyword>;

<keyword> ::= Axiom | Subdiv | Repeat | RX |
   RY | RZ | R | T | S | Priority |
   Epsilon | Rand | "[" | "]";

<filename> ::= {<anychar>-'"'};

<anychar> ::= ? every possible character ?;

<real> ::= ? real number in c++ double notation,
      eg. 1223.324 ?;

<uint> ::= ? unsigned integer ?;
```

List of Figures

1.1. When a ground plan vertex is dragged with the mouse, the windows are automatically distributed. 9
1.2. Because the house is an entity of its own, we can change the texture of all wall tiles with just one drag-and-drop operation. 10
1.3. The iterative process allows rapid adjustments of buildings. On the left side, the original is shown. On the right side, textures, some parameters and the ground plan were modified. All modifications were conducted in about one minute. 11
1.4. On the left side, the original window is shown. The column is combined with the window, and can be flexibly interchanged with other columns. The window has some parameters, like ornament height. On the right side, another column type is used, and ornament parameters are changed, using sliders and drag&drop operations. Those modifications took about one minute. 11
1.5. Visualization of a grammar and the production process 15
1.6. Building generation in context of other operations is shown on the top. Aspects important for visualization are shown below. 18
1.7. Landscape generated using midpoint displacement. The screenshot is from an application written by me for another lecture. 19

2.1. Examples for shapes and subshape relations. Dashed arrows represent coordinate frames. 23
2.2. Examples for labeled shapes and subshape relations. 24
2.3. Production rule with application. 25
2.4. Production rule application using set grammars. 27
2.5. Simplified mughul gardens 28

3.1. Tree generated using L-system. The screen shot is from an application written by me for another lecture. 31
3.2. FASS curve generated using L-system. The screen shot is from an application written by me for another lecture. 33
3.3. Overview on CGA 35
3.4. World coordinate origin and a scope placed in world coordinates. Adapted from [MWH+06]. 37
3.5. From left to right: Queue states after application of production with predecessor `winWall`. The insert positions are not displayed. 43
3.6. Result of split operation 45
3.7. (a) Result of repeat operation (b) Adjustment of repeat sizes 46
3.8. Component splits. From left to right: Side-edge split for corner, top-edge split for fence, vertex split for tower 48
3.9. Alignment of scopes after component splits. Left: Result of a top-vertex split. Top right: Result of a top-edge split. Bottom right: Result of a side-edge split. . 49
3.10. Workflow using component splits. Top left: Start ground plan. Top right: extruded ground plan. Bottom left: side-face component split yields scopes for façades (only one façade is drawn here). Bottom left: Façade is scaled back to 3D. 51
3.11. Left: Without occlusion queries, windows intersect with walls. Right: With occlusion queries. 52
3.12. Application of snap lines. 53
3.13. Automatically extruded groundwork, called groundwork templates. Adapted from [LG06] 57

4.1. The syntax highlighting text editor is shown on the right. On the left side, CGA files are sorted into folders. Below that, the hierarchy of all parsed productions is shown. 61
4.2. Start of a drag-and-drop operation 63
4.3. Result of the drag-and-drop operation 63
4.4. Main user interface, showing parameter modification possibilities. 65
4.5. Before parameter balconyDepth is modified 66
4.6. The dragging of slider balconyDepth modifies the balcony. 66
4.7. Single/Row/Column picking modes 67

4.8. Texture Library. On the left hand side, the hierarchy of texture categories is displayed. Every folder icon represents a category. The user can create, rename and delete categories using the buttons above. Textures can be moved from one category into another with drag-and-drop operations. Every texture belongs to one category. 69
4.9. UML object diagram showing possible derivation 73
4.10. UML class diagram of implicit production hierarchy 75
4.11. Example using VolumeShape 77

5.1. UML class/package diagram of main system 81
5.2. UML class/package diagram of Nodes 82
5.3. UML object diagram of Production node usage 83
5.4. UML class/package diagram of Visitors 85
5.5. UML class diagram of object pools 87
5.6. UML class diagram of implemented flyweight pattern 88

6.1. Building sizes used for the test 93
6.2. Milliseconds per iteration versus triangles for various modes 94
6.3. Iterations per second versus triangles for various modes 95
6.4. Building used for performance profiling 97

List of Tables

6.1. Milliseconds per iteration versus triangles for various modes 92
6.2. Iterations per second versus triangles for various modes 92
6.3. Performance values for different development stages, in percent of total execution time. 97
6.4. Iterations per second versus queries/no queries. 98

Appendix B.

Bibliography

[BBJ+01] P.J. Birch, S.P. Browne, V.J. Jennings, A.M. Day, and D.B. Arnold. Rapid procedural-modelling of architectural structures. In *VAST '01: Proceedings of the 2001 conference on Virtual reality, archeology, and cultural heritage*, pages 187–196, New York, NY, USA, 2001. ACM Press.

[boo07] Boost c++ libraries http://www.boost.org/, 2007.

[Cha89] S.C. Chase. Shapes and shape grammars: from mathematical model to computer implementation. *Environment and Planning B: Planning and Design*, 16(2):215–242, 1989.

[CM05] K.S. Colyar and G.B. Matthews. Procedural modeling of medieval castles. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Posters*, page 8, New York, NY, USA, 2005. ACM Press.

[DB05] J. Döllner and H. Buchholz. Continuous level-of-detail modeling of buildings in 3d city models. In *GIS '05: Proceedings of the 13th annual ACM international workshop on Geographic information systems*, pages 173–181, New York, NY, USA, 2005. ACM Press.

[dSM06] L. Gonzaga da Silveira and S.R. Musse. Real-time generation of populated virtual cities. In *VRST '06: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 155–164, New York, NY, USA, 2006. ACM Press.

[Dua05] J.P. Duarte. Towards the mass customization of housing: the grammar of siza's houses at malagueira. *Environment and Planning B: Planning and Design*, 32(3):347–380, 2005.

[FFC82] A. Fournier, D. Fussell, and L. Carpenter. Computer rendering of stochastic models. *Commun. ACM*, 25(6):371–384, 1982.

[FWB+01] P.A. Flack, J. Willmott, S.P. Browne, D.B. Arnold, and A.M. Day. Scene assembly for large scale urban reconstructions. In *VAST '01: Proceedings of the 2001 conference on Virtual reality, archeology, and cultural heritage*, pages 227–234, New York, NY, USA, 2001. ACM Press.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[Gip99] J. Gips. Computer implementation of shape grammars. *Invited paper, Workshop on Shape Computation, MIT*, 1999.

[GK07] B. Ganster and R. Klein. An integrated framework for procedural modeling. In Mateu Sbert, editor, *Spring Conference on Computer Graphics 2007 (SCCG 2007)*, pages 150–157. Comenius University, Bratislava, April 2007.

[GMB06] K.R. Glass, C. Morkel, and S.D. Bangay. Duplicating road patterns in south african informal settlements using procedural techniques. In *Afrigaph '06: Proceedings of the 4th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 161–169, New York, NY, USA, 2006. ACM Press.

[GS80] J. Gips and G. Stiny. Production systems and grammars: a uniform characterization. *Environment and Planning B: Planning and Design*, 7(4):399–408, 1980.

[Hav05] S. Havemann. *Generative Mesh Modeling. PhD thesis*. TU Braunschweig, 2005.

[KGP05] T. H. Kolbe, G. Gröger, and L. Plümer. Citygml Ű interoperable access to 3d city models. *First International Symposium on Geo-Information for Disaster Management GI4DM*, 2005.

[Kno03] D. Knott. Cinder collision and interference detection in real time using graphics hardware. Master's thesis, UBC, 2003.

[LD03] R. G. Laycock and A. M. Day. Automatically generating large urban environments based on the footprint data of buildings. In *SM '03: Proceedings of the eighth ACM symposium on Solid modeling and applications*, pages 346–351, New York, NY, USA, 2003. ACM Press.

[LG06] M. Larive and V. Gaildrat. Wall grammar for building generation. In *GRAPHITE '06: Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, pages 429–437, New York, NY, USA, 2006. ACM Press.

[LWWF03] T. Lechner, B. Watson, U. Wilensky, and M. Felsen. Procedural city modeling. In *1st Midwestern Graphics Conference*, 2003.

[MH06] R. Miles and K. Hamilton. *Learning UML 2.0*. O'Reilly Media, Inc., 2006.

[MWH+06] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool. Procedural modeling of buildings. *ACM Trans. Graph.*, 25(3):614–623, 2006.

[OMG07] OMG. *Unified Modeling Language: Superstructure, version 2.1.1*. Object Modeling Group, February 2007.

[PF98] U. Piazzalunga and P. Fitzhorn. Note on a three-dimensional shape grammar interpreter. *Environment and Planning B: Planning and Design*, 25(1):11–30, 1998.

[PL96] P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1996.

[PM01a] Y.I.H. Parish and P. Müller. Procedural modeling of cities. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308, New York, NY, USA, 2001. ACM Press.

[PM01b] Y.I.H. Parish and P. Müller. Procedural modeling of cities. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308, New York, NY, USA, 2001. ACM Press.

[PMKL01] P. Prusinkiewicz, L. Mündermann, R. Karwowski, and B. Lane. The use of positional information in the modeling of plants. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 289–300, New York, NY, USA, 2001. ACM Press.

[PMM94] P. Prusinkiewicz, M.J., and Radomír Mêch. Synthetic topiary. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 351–358, New York, NY, USA, 1994. ACM Press.

[pur07] Ibm rational purify http://www.ibm.com/software/awdtools/purify/, 2007.

[SG72] G. Stiny and J. Gips. Shape grammars and the generative specification of painting and sculpture. *Information Processing*, 71:1460–1465, 1972.

[SM80] G. Stiny and W.J. Mitchell. The grammar of paradise: on the generation of mughul gardens. *Environment and Planning B: Planning and Design*, 7(2):209–226, 1980.

[Sti80] G. Stiny. Introduction to shape and shape grammars. *Environment and Planning B: Planning and Design*, 7(3):343–351, 1980.

[Sti82] G. Stiny. Spatial relations and grammars (letter to the editor). *Environment and Planning B: Planning and Design*, 9(1):113–114, 1982.

[WWSR03] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky. Instant architecture. *ACM Transaction on Graphics*, 22(3):669–677, July 2003.

[Zel04] J. Zelsnack. Glsl pseudo-instancing. tech. rep., nvidia corporation. available online at download.nvidia.com/developer/sdk/individual_samples/samples.html., 2004.