Dissertation

Direct Artist Control for Procedural Content Generation of Urban Environments

ausgeführt
zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften

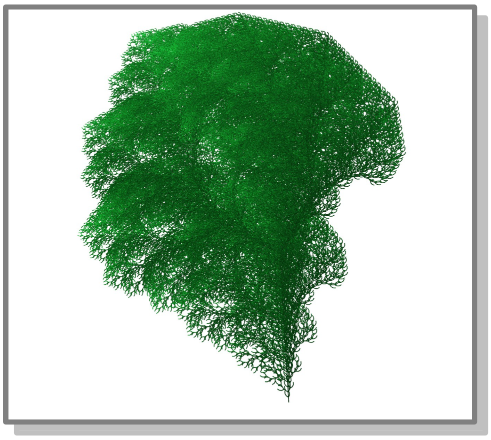
unter der Leitung von
Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer,
Institut für Computergraphik und Algorithmen E186,
eingereicht
an der Technischen Universität Wien,
Fakultät für Technische Naturwissenschaften und Informatik,

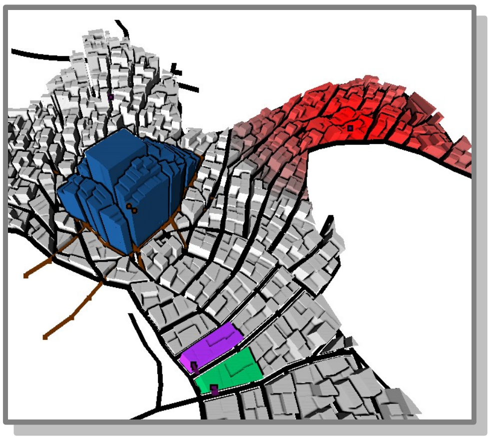
von
Dipl.Ing. Bakk.techn. Markus Lipp,
Matrikelnummer 0125260,
Schönbrunnerstrasse 152/2/5,
A-1120 Wien, Österreich,
geboren am 15. Oktober 1981 in Füssen

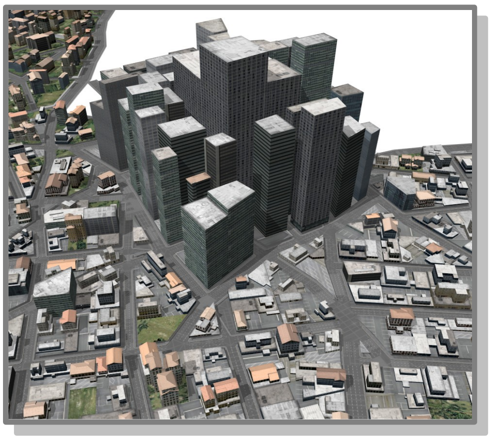
Wien, 18. Oktober 2010.

Direct Artist Control for Procedural Content Generation of Urban Environments

Markus Lipp







Abstract

Creating 3D digital assets of urban environments is a challenging task, requiring a significant amount of manual labor. To automate parts of this process, many procedural modeling methods to automatically create buildings, plants or entire cities were introduced. The main advantage of such methods compared to manual methods is the ability to create large amounts of assets using just a few parameters as input data.

However, the main disadvantage is the difficulty to control or predict the output of such methods. Direct controllability is especially important for artists enabling them to model the output to their vision or requirements. Therefore, the main goal of this thesis is combining the direct control provided by manual methods with the power of procedural modeling.

To achieve this, several new methods and paradigms bringing direct and visual artist control to procedural generation of urban environments are contributed in this thesis. These include a method enabling a visual design process for building grammars, as well as methods providing direct artist control for architecture generation algorithms. To model whole cities, we introduce a layering system for urban layouts based on graph cut, contributing the ability to perform direct and persistent changes to a procedurally generated city. Additionally, the concept of anchored assignments is introduced, enabling direct control of parameter distributions on cities. Finally, as real-time performance of generation algorithms is paramount if they are to be used in a production setting, we introduce an algorithm able to parallelize the work of L-system generation to thousands of processors.

Kurzfassung

Die Erstellung von digitalen 3D Inhalten für virtuelle Städte ist eine große Herausforderung und beinhaltet viel manuelle Arbeit. Deshalb wurden viele prozedurale Methoden zur automatischen Erstellung von Gebäuden, Pflanzen oder kompletten Städten vorgestellt. Der große Vorteil solcher Methoden ist die Möglichkeit, anhand von wenigen Parametern als Eingabedaten große Mengen von digitalen Inhalten zu erstellen.

Allerdings haben diese Methoden auch einen großen Nachteil: Im Vergleich zu manuellen Methoden sind die resultierenden Inhalte schwer vorherzusehen oder zu beeinflussen. Die Möglichkeit, direkte Beeinflussung auf das Resultat auszuüben, ist besonders für Designer wichtig, damit diese ihre Vision oder Anforderungen umsetzen können. Deshalb ist das Hauptziel dieser Dissertation, die direkte Beeinflussbarkeit manueller Methoden mit der Leistungsfähigkeit prozeduralen Modellierens zu kombinieren.

Um direkte und visuelle Beeinflussung von prozeduralen Methoden zur Erstellung von virtuellen Städten zu ermöglichen, führen wir mehrere neue Methoden und Paradigmen ein. Zuerst stellen wir eine Methode zum visuellen Design von Grammatiken für Architekturerstellung vor, des weiteren werden Methoden zur direkten Beeinflussung von Methoden zur Architekturgenerierung dargelegt. Um ganze Städte zu modellieren, stellen wir ein System basierend auf mehreren Ebenen von Stadtteilen unter Benutzung des Graphcut Algorithmus vor. Dies ermöglicht direkte Beeinflussung von prozedural erstellten Städten. Des weiteren wird ein Konzept zur direkten Definition von Parametern für Städte vorgestellt. Da interaktive Geschwindigkeit von prozeduralen Algorithmen nötig ist wenn diese in einem Projekt eingesetzt werden sollen, stellen wir auch einen Algorithmus zur Parallelisierung von L-Systemen auf über tausend Prozessoren vor.

Contents

1

Introduction

Virtual models of urban environments, consisting of streets, buildings and plants, are required in many applications. For example, they can be used in movies or computer games for *entertainment* purposes. Here, convincing virtual models of cities are often required for special effects or interactive environments. For movies, it is especially important that the models are of high visual quality. Another application is *urban planing*: Visualizing planned buildings or districts using virtual models can help in decision making during an urban planing process, allowing an estimation of the impact of planned buildings on the cityscape. Finally, they can be used for *cultural heritage* applications. Virtual models of historic cities can be used for education and for gaining insight into long gone cultures. In Figure 1.1 those applications are illustrated.

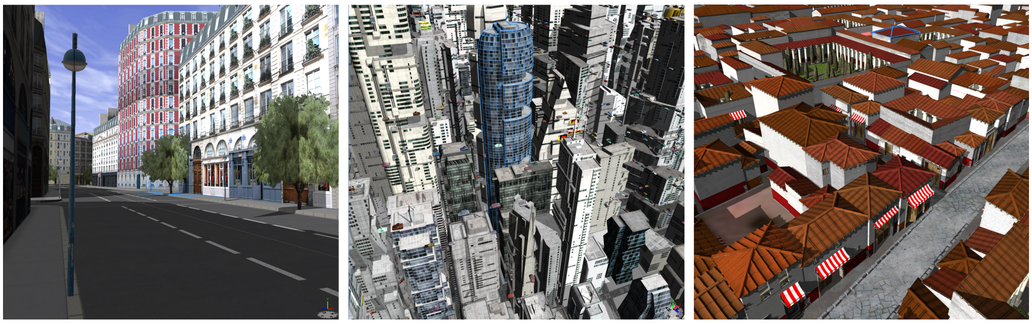


Figure 1.1: *Applications of virtual urban models. Left: In this urban planning scenario, the impact of the planned red building on the cityscape can be evaluated. Middle: Futuristic city created for a computer game. Right: A virtual model of Pompeii for cultural heritage applications. Images generated with Procedural Inc. Cityengine [Pro10].*

Creating such virtual models is a challenging task. A traditional approach is to manually model the numerous virtual objects occurring in an urban environment, using polygonal modeling tools like Autodesk Maya$^{\text{TM}}$. One strength of this approach is the full controllability an artist has over every aspect in the city, as essentially everything is hand crafted. The obvious disadvantage is the huge amount of time, resources and manpower required for creating whole cities.

One reason why this is such a costly process is that reusing assets for different objects in a city can only be performed on a low level, by copying or instancing parts of a model. In traditional tools, there is no way to extract higher level aspects of objects in order to reuse them in a different part of the city. Also, as objects are successively modeled, it is difficult to make small changes to a city layout after the buildings were created. For example, when a street is moved, the buildings need to be manually updated.

Therefore, the main idea to reduce the costs of modeling urban environments is to use *procedural* modeling techniques, which are able to automatically generate models out of a higher level description, and can regenerate objects after parameter changes. We will look at such methods in the next section.

1.1 Procedural Modeling

To automate parts of the urban environment modeling process, many procedural modeling techniques were introduced. Based on the underlying methodology, these techniques can be roughly classified into three groups:

- Grammar based: Given a set of rules, a start symbol and parameters, the start symbol is iteratively replaced with other symbols based on the rules.
- Simulation based: Here, the idea is to simulate a natural growth or urban development process using an abstract model of the process.
- Example based: The main idea is to employ parts of databases, e.g. geographic information systems (GIS), or images as examples for content generation. This data can either be exploited by extracting parameters for a grammar- or simulation-based method, or more directly by combining multiple smaller parts of this data to create new content.

All these methods share a general input, procedural generation and output model, as illustrated in Figure 1.2. This underlying concept accounts for the main strength of procedural generation: Out of a relatively small set of input data, procedural generation can automatically create large amounts of output. Essentially, the input data is a high-level description of the models to be generated. Also, changes to the input data followed by a regeneration are simple and inexpensive.

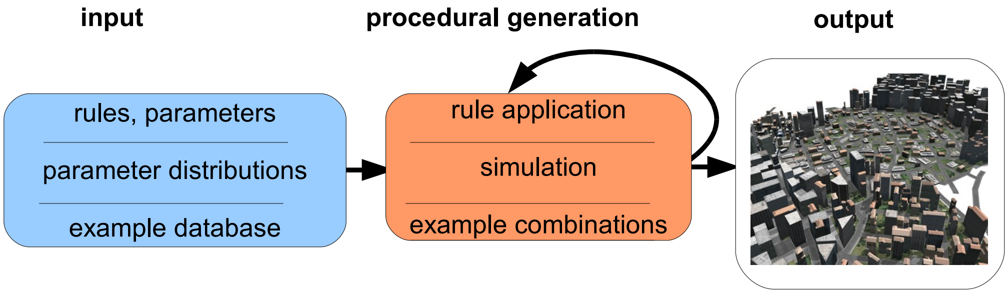


Figure 1.2: *Conceptual overview of procedural methods: A possibly iterative procedural generation algorithm creates output assets from input data.*

Object types Apart from the underlying methodology, procedural algorithms can also be classified by the types of virtual objects they are able to create. There is no one-to-one mapping between those classifications, as some methods have been applied to different object types.

We will now explain the main types occurring in an urban environment: For *plants* mainly grammar-based approaches, such as L-systems are employed. To generate *buildings* and general *architecture*, grammar-based approaches and example-based approaches using images were introduced. A wide variety of methods have been applied to create *urban layouts*, which we define as street networks with parcels and parameter distributions. The methods include simulation-based approaches using urban development models, example-based approaches using GIS data, and grammar-based approaches using L-systems.

Of course there are much more object types, for example landscapes or textures. In order to limit the scope of this thesis to a reasonable area, we assume these objects already exist.

Procedural modeling pipeline Now that we introduced the methods and object types, the question arises how the methods can work together to create a complete virtual urban environment. The main idea is to use a *pipeline*, chaining together the different methods. This has been proposed by Pascal and Müller [PM01]. We will now provide a conceptual introduction to their pipeline, a more detailed explanation will be given in Section 2.3.1.

(1) A landscape combined with parameter distributions of desired building types and properties is provided by the artist. (2) A procedural street generation algorithm creates a street network using provided parameters and the distributions from step 1. (3) Parcels are extracted from the street network. (4) Finally, for every parcel and every street, a specific procedural algorithm for geometry generation is employed. The algorithm can create street geometry, generate and distribute plants, or create buildings. It takes

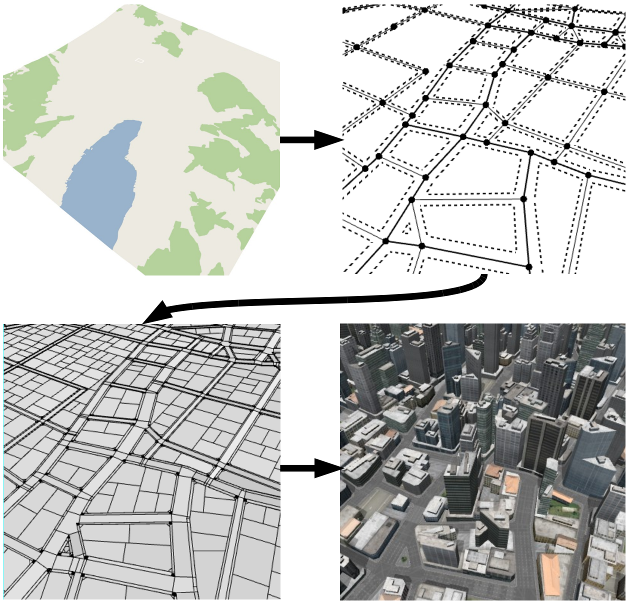


Figure 1.3: *Conceptual procedural pipeline for urban envrionment generation. Starting from a landscape, initially the street network, then the parcels, and finally the buildings, plants and streets are created.*

the parameters provided in step 1 as well as the street or parcel geometry as input. The complete pipeline is visualized in Figure 1.3.

1.2 Artist Control for Procedural Modling

Until now, we only mentioned the advantages and possibilities of procedural methods. However, those methods also have one significant challenge: *Where and how can artists control the method to generate an output that corresponds to their vision or requirements?* For manual modeling, direct control of the output is always provided. If we want to combine the direct control of manual modeling with the power of procedural modeling, we have to look at solutions to bring direct control to procedural methods. *This is the main goal of this thesis.* In previous work this has often been neglected, as the emphasis was more on the generation algorithms.

Let us first look at why direct control so important for the application areas mentioned in the beginning. Then we will examine the potential types of artist control, including direct control. Finally, possible visual interfaces for artist control will be explained.

1.2.1 Importance of artist control

For movies and computer games, there is always a specific vision of a director or designer how the result should look like. While procedural algorithms can quickly create large amounts of content, there can be the desire to directly control details of this output to exactly correspond to this vision. When we talked with artists from a game company about procedural methods, they said it is imperative for them to have direct control if they are to employ procedural methods.

For urban planing, the ability to perform interactive modifications of planned areas can be a powerful tool in order to explore new ideas or alternative designs. Those modifications need to be direct in order to convey the architect's vision.

In cultural heritage applications, it is important to be able to incorporate new findings directly into the virtual model. Also, a scientist should be able to incorporate domain knowledge not encoded into the procedural algorithm into the model.

1.2.2 Types of Artist Control

Essentially, there are three ways an artist can control procedural content creation, as visualized in Figure 1.4. (1) The artist can change the input data, for example by modifying some rules. This has a global effect, as all subsequently generated assets will use this new input, and only provides indirect control over the output. (2) The generation process can be steered locally, for example by specifying a population distribution of a city. This is still an indirect approach. Additionally, local steering does *not* ensure that the effect on the output is also local. For example reducing population density locally could cause the generation algorithm to remove a large highway that was present before. However, it is possible to design algorithms that ensure local steering has local effect, therefore we classify local steering as having both local and global effect. (3) The artist can directly and locally change the output. As this provides the maximum control, we will focus our thesis on this type. Let us look at direct changes in more detail.

Direct control While modifying the output directly provides maximum control, there is one significant challenge, best explained with an example illustrated in Figure 1.5: Suppose the artist directly changed one window of a building. Afterwards the artist may decide to change one input parameter of the procedural generation, for example the width of the windows. To account for this global change, the building has to be deleted and regenerated, as

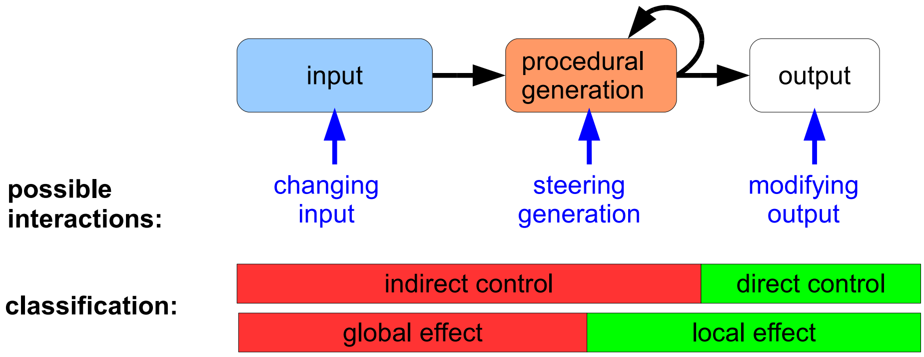


Figure 1.4: *Classification of possible interactions in a procedural generation system.*

changing the window width may change the amount of windows. However, during regeneration the system does not have any knowledge of the modified window, as this is not encoded in the input. Therefore the direct modification is lost.

In most cases this is probably not what an artist intended: The system should be able to retain direct local changes even when a global change is performed afterwards. This would combine the data amplification power of procedural modeling with the direct control provided by standard 3d modeling tools. Most previous work only focuses on indirect and global control. Therefore, one goal of this thesis is to find solutions to allow *persistent direct control* in procedural algorithms.

1.2.3 Visual Interfaces for Artist Control

Apart from the question where to apply artist control, it is also important to look at how those control possibilities are presented to the artist.

Input In grammar-based approaches, mostly a text-based definition of the rules is used as input. This can be unintuitive for an artist, a visual representation of the rules would be preferable. Therefore one goal of our work is to find a visual system for grammar creation. For simulation- and example-based approaches the visual representation of the input is easier. For instance, in an example-based approach the examples can simply be rendered.

Steering generation Instead of text-based descriptions, visual control structures are more suited to steer the generation. Different visual controls have been introduced in the context of L-systems, as will be pointed out

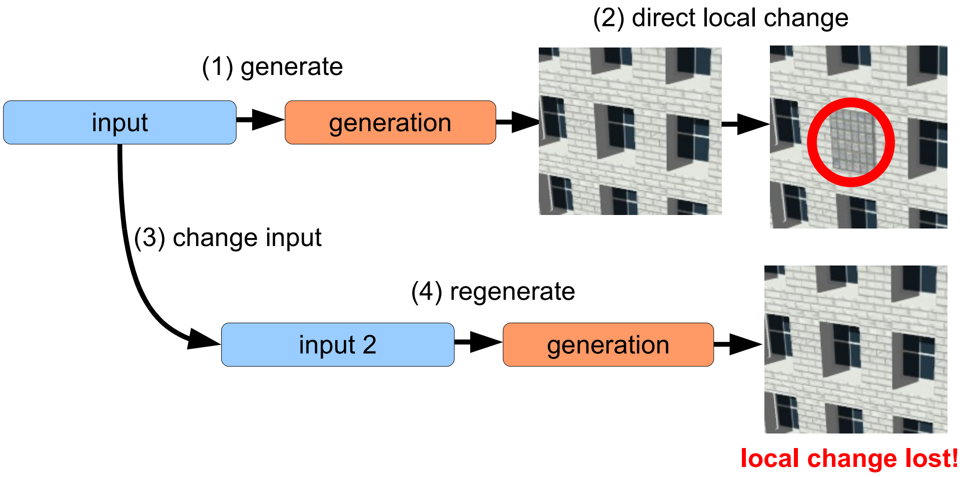


Figure 1.5: *Illustration of the* persistence *problem. The artist interactions are numbered. After an input change the previous local change is lost.*

in the related work section. Also, parameter distributions for cities can be represented as color coded images.

Modifying output One way to modify the output is to use traditional tools like Autodesk Maya$^{\mathrm{TM}}$. However, instead of the low-level geometry editing that such tools provide, it would be beneficial to have high level editing operators tailored to the specific problem domain. For example, when editing buildings it should be possible to use drag and drop operations to change a window type. When editing urban layouts, tools to move complete streets would be beneficial. We introduce such tools in Chapter and 3 and 4.

1.3 Real-Time Performance

To truly exploit the power of procedural modeling combined with direct control, it is imperative that direct changes can be performed interactively in real-time. Local or global changes should immediately reflect on the rendered result. Therefore, another goal of this thesis is speeding up existing procedural algorithms to make real-time interactions feasible.

As it would be impossible to cover performance improvements for all procedural methods in the scope of this thesis, we will just focus on accelerating L-system based approaches. We see this as a first step towards accelerating other grammar based approaches as well.

1.4 Dissertation thesis

This work focuses on providing direct artist control for procedural content creation methods of urban environments.

> The main thesis of this work is that it is possible to add persistent direct control to procedural methods, and that the resulting methods enable novel ways of visual interaction. Further we claim that it is possible to achieve real-time performance for such methods using parallelization and other techniques.

We will underline this statement by introducing several methods for direct control, and by presenting feedback to those methods from artists of a game company. Further, we will provide performance evaluations of our methods and introduce a method specifically targeted at parallelizing L-systems.

1.5 Contributions

First, we introduce a visual interface for grammar creation as well as a direct control solution for architecture generation. Then, we introduce direct control solutions for city generation. Finally, we introduce a parallel algorithm for fast generation of multiple L-systems.

Direct Control for Procedural Architecture We introduce a real-time interactive visual editing paradigm for shape grammars, allowing the creation of rulebases from scratch without text file editing. In previous work, shape-grammar based procedural techniques were successfully applied to the creation of architectural models. However, those methods are text based, and may therefore be difficult to use for artists with little computer science background. Therefore the goal was to enable a visual workflow combining the power of shape grammars with traditional modeling techniques. We extend previous shape grammar approaches by providing direct and persistent local control over the generated instances, avoiding the combinatorial explosion of grammar rules for modifications that should not affect all instances. The resulting visual editor is flexible: All elements of a complex state-of-the-art grammar can be created and modified visually. The results [LWW08] have been published in:

- Markus Lipp, Peter Wonka, and Michael Wimmer. Interactive visual editing of grammars for procedural architecture. ACM Transactions on Graphics, 27(3):102:1-10, 2008. Article No. 102.

Direct Artist Control for Procedural City Layouts We present new solutions for the interactive modeling of city layouts that combine the power of procedural modeling with the flexibility of manual modeling. Procedural modeling enables us to quickly generate large city layouts, while manual modeling allows us to hand-craft every aspect of a city. We introduce transformation and merging operators for both topology preserving and topology changing transformations based on graph cuts. In combination with a layering system, this allows intuitive manipulation of urban layouts using operations such as drag and drop, translation, rotation etc. In contrast to previous work, these operations always generate valid, i.e., intersection-free layouts. Furthermore, anchored assignments make sure that modifications are persistent even if the whole urban layout is regenerated, for example following a global parameter change in the procedural definition. The results are currently under review.

Parallel generation of multiple L-systems We introduce a solution to compute L-systems on parallel architectures like GPUs and multi-core CPUs. Our solution can split the derivation of the L-system as well as the interpretation and geometry generation into thousands of threads running in parallel. We introduce a highly parallel algorithm for L-system evaluation that works on arbitrary L-systems, including parametric productions, context sensitive productions, stochastic production selection, and productions with side effects. This algorithm is further extended to allow evaluation of multiple independent L-systems in parallel. In contrast to previous work, we directly interpret the productions defined in plain-text, without requiring any compilation or transformation step (e.g., into shaders). Our algorithm is efficient in the sense that it requires no explicit inter-thread communication or atomic operations, and is thus completely lock free.

The results [LWW10] have been published in:

- Markus Lipp, Peter Wonka, and Michael Wimmer. Parallel generation of multiple l-systems. Computers Graphics, 34(5):585-593, 2010. CAD/GRAPHICS 2009; Extended papers from the 2009 Sketch-Based Interfaces and Modeling conference; Vision Modeling Visualization.

1.6 Overview

The remainder of this thesis is structured as follows: In Chapter 2 the related work for procedural content creation and parallelizing of L-systems is presented. Chapter 3 focuses on direct visual control for procedural architecture. In Chapter 4 a method for city-wide direct control of streets and parcels using a layering system is presented. Chapter 5 details algorithms for parallel generation of L-systems. A summary of the thesis is provided in Chapter 6.

2

Related Work

In this chapter we summarize the related work of our research. First, methods for procedural generation with corresponding concepts for artistic control are introduced. Section 2.1 focuses on plants, Section 2.2 on architecture and Section 2.3 on urban layouts. Then, methods for parallelizing L-systems will be shown in Section 2.4.

An extensive review of procedural urban modeling methods can be found in a recent survey paper by Vanegas et al. [VAW$^+$10]. While we cover a similar area, our focus is on visual artist control.

2.1 Plants

Plants are mainly generated using L-systems. L-systems were introduced by Prusinkiewicz and Lindenmayer [PL96]. While they are mainly employed to generate plants, they were also used to generate buildings [PM01]. One example plant is shown in Figure 2.1.

We will now provide an overview of the formalism as introduced by Prusinkiewicz and Lindenmayer [PL96]: Parametric L-systems operate on *parametric words*, which are strings of *modules* consisting of *letters* with associated *actual parameters*. An L-system consists of a parametric word ω called the *axiom*, and a set of productions describing how the current word is transformed. A production consists of a letter possibly combined with *formal parameters*, called the *predecessor* and a *successor*. The successor consists of a list of letters, where each letter can have multiple *arithmetic expressions* containing formal parameters. Formal parameters can be global or local to one production rule. The real-valued *actual parameters* appearing in the words are calculated from the arithmetic expressions of *formal parameters*. The predecessor can also consist of several letters, in which case the L-system is called context sensitive [PL96].

In the following example, F, A, and B are the letters defining modules,

Figure 2.1: *An example 3D plant generated using L-systems. Image courtesy of Prusinkiewicz and Lindenmayer [PL96].*

g_i are global parameters, l is a local parameter, and the arrow separates the predecessor from successor:

$$F(l) \rightarrow A(l * g_1)[B(l + g_2)]$$

To actually generate geometry, two distinct phases are performed: A *derivation* phase generating a string of modules, and an *interpretation* phase in which the string of modules is interpreted in order to generate geometry.

Derivation The derivation starts from the axiom. For every module contained in the axiom, a *matching* production is searched. A production matches a module m if the letter of the predecessor matches the module letter, and the number of actual parameters in the module equals the number of formal parameters in the production. We then *apply* the matching production to the module: First, for every module in the successor, we calculate the actual real-valued parameters from the arithmetic expression of the formal parameters. Then we *rewrite* the module m with the modules of the successor. One *iteration* consists in rewriting all modules in the string *in parallel* using matching productions [PL96]. A user-defined amount of iterations is performed in order to get the final string of modules.

Interpretation The interpretation is performed *serially* from the start of the string, performing modifications of a *turtle state* based on predefined *turtle commands* associated with specific letters [PL96]. The turtle state represents the position and orientation of a virtual turtle. This state can be represented with a 4x4 matrix. The turtle commands associated to letters modify the turtle state, for example 'F' moves the turtle forward while drawing a line, or '+' rotates the turtle. Most of these turtle commands can also be expressed by a 4x4 matrix. A notable exception are the commands '[' and ']', which push and pop the turtle state on a stack, allowing the creation of *branching* (also called *bracketed*) L-systems [PL96].

Artist control for L-systems Several extensions were proposed to L-systems, mainly applicable to plants: Prusinkiewicz et al. [PMM94] show how to restrict the growth process of plants to a user-defined volume by introducing pruning, enabling the creation of synthetic topiaries. An example output is shown in Figure 2.2. To achieve this, a query module is introduced. After each derivation step this module allows querying the position or orientation of the turtle. This information can be used for intersection tests with a bounding object.

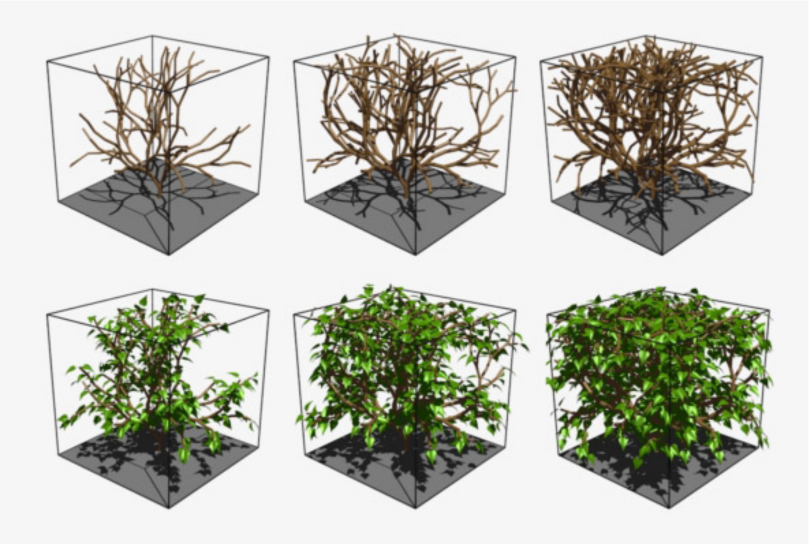


Figure 2.2: *Synthetic topiaries created using pruning to a bounding box. Image courtesy of Prusinkiewicz et al. [PMM94].*

To actually create plants reacting to pruning, the authors specify an L-system using this module. When collisions are found, context sensitive rules are employed to send signals to higher level branches, allowing them to adapt to the collision by bending. This collision response was later extended to simulate plants interacting with their environment, competing for natural resources, using so-called open L-systems [MP96].

Later, the idea of the query module was generalized by introducing a function module [PMKL01]. This module allows querying an arbitrary, user-defined function during interpretation, using the relative turtle position as parameter. By modifying the functions, an artist can influence the shape of plants, as shown in Figure 2.3. Boudon et al. [BPF$^+$03] combined bounding shapes and function modules, and introduced a multiscale representation of plants in order to minimize the total number of parameters needed to specify a plant.

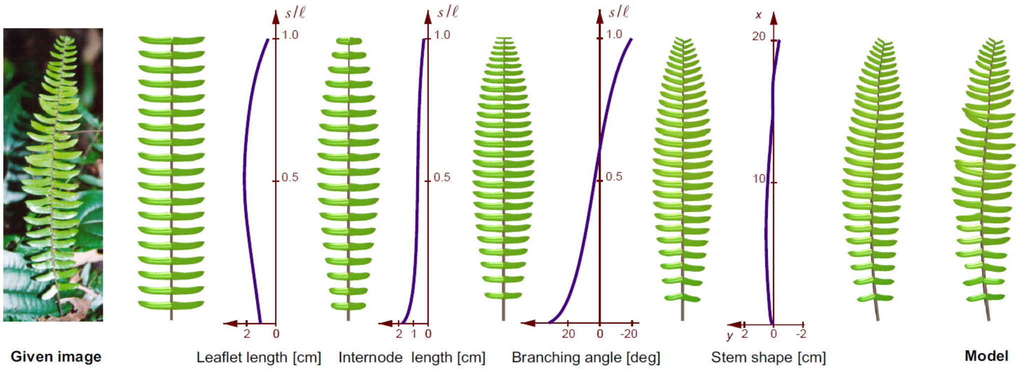


Figure 2.3: *Using function modules, the relative turtle position can be used to query a user-defined function. This allows direct control of plant shapes. Image courtesy of Prusinkiewicz et al. [PMKL01].*

L-systems were also used to create street layouts, using so-called extended L-systems [PM01]. They will be introduced in in Section 2.3.1.

Visual modeling of L-system rules To help artists in creating L-system rules, which are generally text-based, a visual system to model grammars was proposed by Lintermann and Deussen [LD99]. They introduced visual components specifically targeted at plant modeling, for example leafs and branches, that can be connected in a graph. Those components are very specific to plants and can not be used to model other objects like buildings. One example graph is shown in Figure 2.5.

Analysis Both the query and function module share the same general concept to enable artistic control, as visualized in Figure 2.4: First, the grammar rules need to be specifically crafted using those modules. Rewriting an existing L-system to use those modules is a non-trivial task, which is the main drawback of those methods. Second, a control structure external to the L-system is specified and queried during the derivation by the modules. Third, to influence the plant, the artist does not directly control the resulting plant, instead the control structures are modified and the plant reacts as specified in the rules.

To sum it up, the main disadvantage is that the query modules need to be tightly integrated into the grammar, additionally the control is still indirect. In our work we try to alleviate this by enabling direct control without query modules, as will be shown in Section 3.

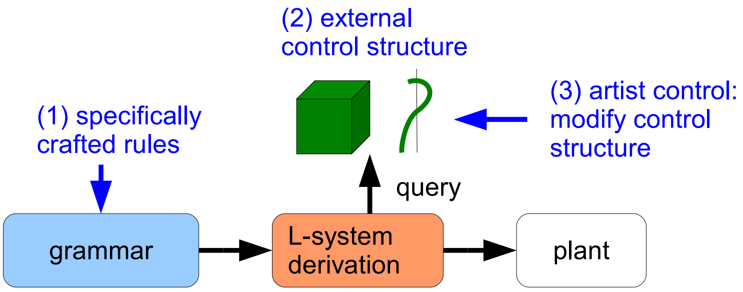


Figure 2.4: *Conceptual view of artistic control for L-systems.*

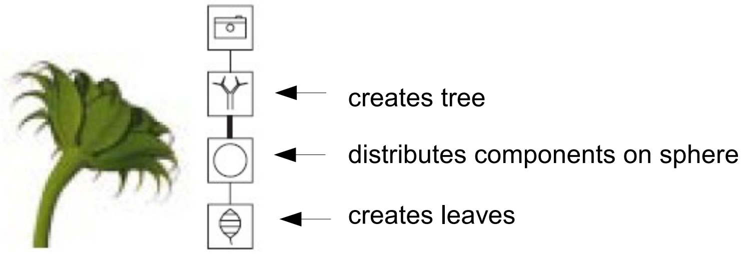


Figure 2.5: *Components can be connected by an artist to create plants. Image courtesy of Lintermann and Deussen [LD99], annotations added.*

2.2 Architecture

In this section we will introduce concepts for procedural generation of architecture, starting from early shape grammars, continuing with split grammars and finally providing details on alternative approaches.

2.2.1 Shape grammars

Shape grammar were pioneered by Stiny and Gips [SG72, Sti80]. In contrast to L-systems, shape grammars operate on lines and points instead of symbols.

Let us provide a condensed version of the definitions by Stiny [Sti80]. We left out details like labeled and parametric shapes.

A shape is an arrangement of lines. Shape s_1 is a subshape of shape s_2 (denoted $s_1 \leq s_2$) when every line of s_1 is also in s_2. A Euclidean transformation of shape s is denoted as $\tau(s)$. Shape s_1 is geometrically similar to shape s_2 if there is a transformation τ such that $\tau(s_1) = s_2$.

A shape grammar consists of a set of shapes S, a set of shape rules R of the form $\alpha \to \beta | \alpha \in S^+, \beta \in S^*$ and an initial shape $s \in S^+$. A shape rule $\alpha \to \beta$ *applies* to a shape γ when there is a transformation $\tau | \tau(\alpha) \leq \beta$. The shape produced by applying $\alpha \to \beta$ to γ is $(\gamma - \tau(\alpha)) + \tau(\beta)$, where $+$ and $-$ represent Boolean union respectively Boolean difference. New shapes are generated by iteratively applying shape rules starting from the initial shapes. For better understanding, an example shape grammar is shown in Figure 2.6.

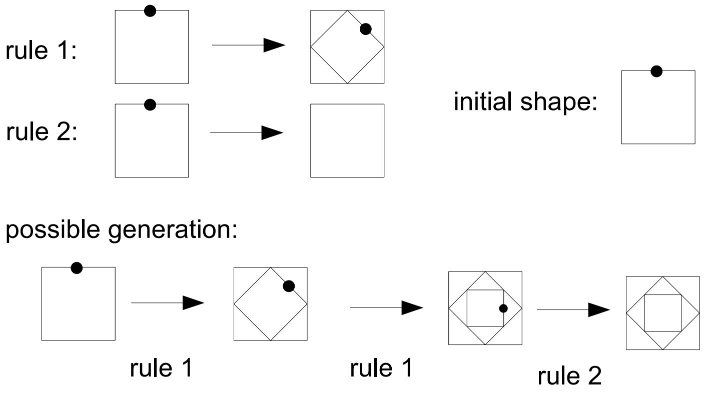


Figure 2.6: *Two shape rules and a possible outcome of the generation are illustrated here. The points represent labels. Note that this grammar can create an infinite amount of generation results. Figure based on illustration from Stiny [Sti80].*

Implementation challenges One major challenge when implementing a shape grammar interpreter is finding a transformation $\tau | \tau(\alpha) \leq \beta$, called the subshape problem [Cha89]. This is essentially a pattern matching problem: For every rule, transformations of α have to be searched in the current shape. This is computationally very expensive, and recently it was even proven that the parametric subshape recognition problem is NP-hard [YKG09]. To simplify the subshape problem, set grammars were introduced [Sti82]. Essentially, instead of working directly on lines, set grammars combine multiple lines to a set, reducing the search space for rule applications. Most existing implementations circumnavigate the general subshape problem either by restricting the grammar to set grammars or other restrictions such as only supporting rotations of 90 degrees [Cha89]. To our knowledge, currently there exists no automatic 3d shape grammar interpreter without such restrictions.

Another challenge is *emergent* behavior of shape grammars. This essentially refers to the possibility of obtaining surprising results not intended by the artist [Cha89]. On the one hand, this could be interpreted as a strength of shape grammars, as emergence can lead to creative new results. On the other hand, emergence hinders direct artist control, as unintended results are possible.

Applications Many shape grammars have been proposed for various applications. These include paintings [SG72], Palladian houses [SM78], Mughul gardens [SM80] (shown in Figure 2.7) or mass housing [Dua05].

It is important to note that those grammars were not targeted at the creation of visually convincing models for movies or computer games (including textures or ornaments), instead accurate ground plans were produced.

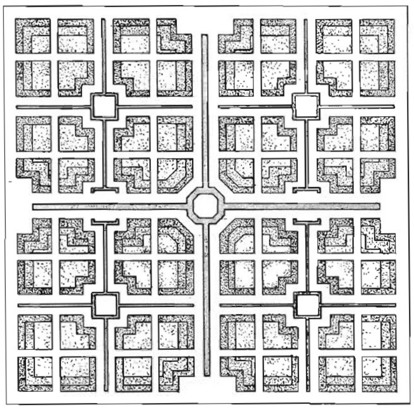


Figure 2.7: *Mughul garden created using shape grammars. Image courtesy of Stiny and Mitchell [SM80].*

2.2.2 Split grammars

The first method to actually generate visually convincing models with high geometric façade detail was *Instant Architecture* [WWSR03]. This method introduced split grammars, which are based on set grammars. The basic formalism is similar to set grammars, however there are some important contributions: (a) They introduce a split command, defined as the decomposition of a basic shapes into other shapes. (b) A parameter matching system is introduced to find applicable rules. This simplifies the sub-shape problem. (c) A control grammar is introduced to provide control over spatial distributions, essentially circumnavigating the emergence problem. An example derivation is shown in Figure 2.8

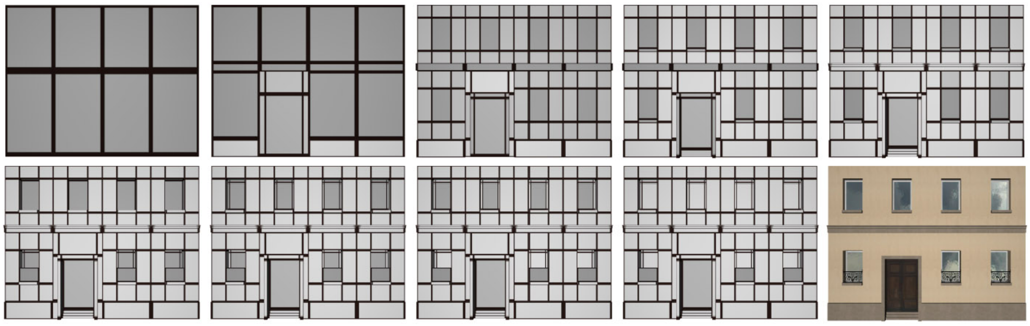


Figure 2.8: *Example derivation of a split grammar. Image courtesy of Wonka et al. [WWSR03].*

A main advantage of split grammars is that facades with high detail can be generated using a very simple language. The main disadvantage is that the control is very indirect for an artist: The parameters for the matching system have to be carefully set, and a fine-tuned control grammar is vital to get the desired result.

CGA shape To improve on split grammars, *CGA shape* was introduced [MWH$^+$06], extending split grammars in the following ways: They were the first to actually *define the syntax* of split commands. They introduced the *component split*, which allows reducing the dimensionality of the current scope. Additionally, mass modeling was introduced to create more complex buildings shells. Finally, the control grammar and attribute matching were removed, every rule has a probability instead. CGA shape is the basis for the commercial CityEngine [Pro10], and has inspired much subsequent work. Let us therefore provide a condensed definition of CGA shape:

A *shape* consists of a symbol, geometry and attributes. Attributes include an oriented bounding box, called the scope. Production rules replace one shape with other shapes. The production process starts with an arbitrary configuration of shapes, called the axiom. Iteratively, shapes are replaced according to rules.

There are scope transformation rules for translation, rotation, scaling. Also push and pop of the scope on a stack is supported. The basic *split* rule splits the current scope along a specified axis with user-defined split widths. The shapes placed on the split positions are also user defined. A component split allows to split into shapes of lesser dimensions.

Mass models can be created either by extruding building lots, or importing 3d geometry. In order to prevent occlusions of facade features, tests querying for occlusions can be employed in rules. An example output is shown in Figure 2.9.

Figure 2.9: *City created using 190 CGA shape rules. Image courtesy of Müller et al. [MWH+06].*

CGA shape is considered state-of-the-art for procedural building generation. However, it has two disadvantages: First, the rules are completely text based, making input definition unintuitive for an artist. Second, no persistent direct control of the output is supported. We will provide solutions to both problems in this thesis.

2.2.3 Alternative approaches

Here we will introduce methods that are not directly grammar based, but still enable some kind of visual artist control for procedural architecture generation.

Image based approaches Müller et al. introduced a method able to extract a 3d model and shape grammar rules from a single rectified image of a facade [MZWG07]: (1) Symmetry detection is used to create an irreducible facade (IF) image. An IF is a summary of the facade, with all symmetries of building tiles removed. (2) An edge detection algorithm then subdivides the

IF into floors and tiles. (3) By expanding the IF again to the full facade, the subdivison is known for the full facade. (4) Each tile is further subdivided into windows and ornaments, first locally, then the local results are globally synchronized. (5) Predefined 3d meshes of windows and ornaments are automatically matched to the subdivided elements. (6) The output thus far does not have depth information. A user can manually add it. Finally, shape grammar rules are automatically extracted from the subdivision tree. An example is shon in Figure 2.10.

Figure 2.10: *3d facade extracted from a single image. The extracted shape allows flexible resizing of the output. Image courtesy of Müller et al. [MZWG07].*

For the previously introduced method, images need to be free of occluding objects such as traffic lights and wires. Musialski et al. therefore introduced a method to automatically remove those occluders by exploiting symmetries in the facades [MWR$^+$09].

Xiao et al. introduced a method taking multiple images of the facade as input [XFT$^+$08]. They claim that the approach from Müller et al. [MZWG07] only works well for repetitive facades. Their results show also irregular facades, as shown in Figure 2.11. However, they do not extract shape grammars for the facades, so resizing is not possible. The general algorithm works as follows, most steps also allow require the user to perform fine-tuning: (1) Structure from motion is used to calculate a 3d point cloud of the facades. (2) Assuming the facade is a flat rectangle, the images are projected onto the facade. Occluded pixels in an image are detected and replaced with pixels from other images. (3) The facade is subdivided. (4) Finally, depth values are calculated from the point cloud. Xiao et al. further extended their algorithm to enable geometry generation for areas of cities [XFZ$^+$09].

Generative modeling Havemann introduced the Generative Modeling Language (GML) [Hav05], a concise definition was provided by Berndt et al. [BFH05]. GML is a is stack-based programming language, similar to

Figure 2.11: *3d facades extracted from multiple images. On the right, an irregular facade is shown. Image courtesy of Xiao et al. [XFT$^+$08].*

PostScript. An example output is shown in Figure 2.12. The GML interpreter is very simple, but relies on a large library of operators. All functionality of the GML comes from the operators, organized in several libraries [BFH05]. Probably the most important is the `CBRep` library providing access to the combined BRep (CBRep) meshes introduced in their paper. CBPrep is a multiresolution data structure, operating on input meshes by subdividing their faces [BFH05].

Advantages of GML include the compact size for model representation and the possibility to specify semantic level of detail. As a disadvantage, the indirect geometry specification is probably unintuitive for an artist.

Figure 2.12: *Gothic windows created using GML. Image courtesy of Berndt et al. [BFH05].*

Functional Modeling Paoluzzi et al. introduced PLASM (acronym for "the Programming LAnguage for Solid Modeling") [PPV95]. In contrast to GML it is based on a functional language instead of a stack-based one. Any PLASM function can produce infinitely different geometric models with some common structure, based on parameters [PPV95]. Because PLASM is based on a functional language, different PLASM functions can be easily combined. By using Boolean operations, the result of all combinations can be guaranteed to be geometrically valid. Originally, modeling in PLASM was completely text-based. A newer paper introduces a mapping of PLASM functions to visual symbols [MBP05]. An example is shown in Figure 2.13. While this may make definitions PLASM programs easier, it is still a very indirect method of geometry specification.

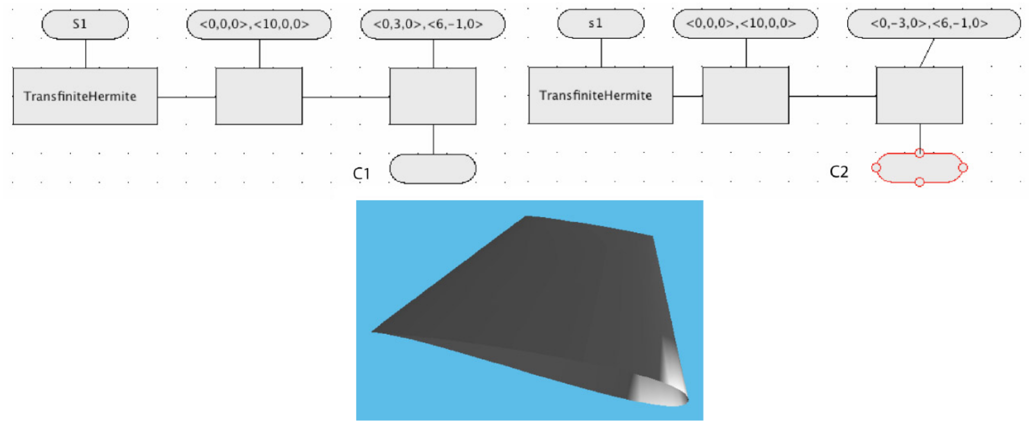


Figure 2.13: *One the top, an excerpt of visual symbols representing a functional language is shown. On the bottom, a wing profile created using such a language is shown. Image courtesy of Milicchio et al. [MBP05].*

Similarly to GML, the advantage of PLASM is a very compact representation of parametrized models. Further advantages include modularity due to the functional approach. As a disadvantage, simmilar to GML, the indirect specification of 3d objects using a programming language can be unintuitive.

Visual modeling language Conceptually similar to the previously introduced visual language for PLASM, Ganster and Klein [GK07] introduced visual symbols for procedural modeling. Instead of a functional language, the underlying language is procedural. Multiple visual symbols are introduced and can be connected in a graph. The connection edges define the order of operations. Symbols include functionality for texture loading, geometry creatuion (spheres, cylinders, tree stems), transformation, and flow control (for, while). An artist can visually connect the symbols to create simple trees, buildings and landscapes, as shown in Figure 2.14. Advantages of this approach include applicability to a variety of modeling subjects. As the visual language is essentially a mapping of low-level programming structures to symbols, the graphs for simple scenes can become large and difficult to manage, for example the landscape generated in Figure 2.14 contains 32 symbols with 8 interleaved `for` loops.

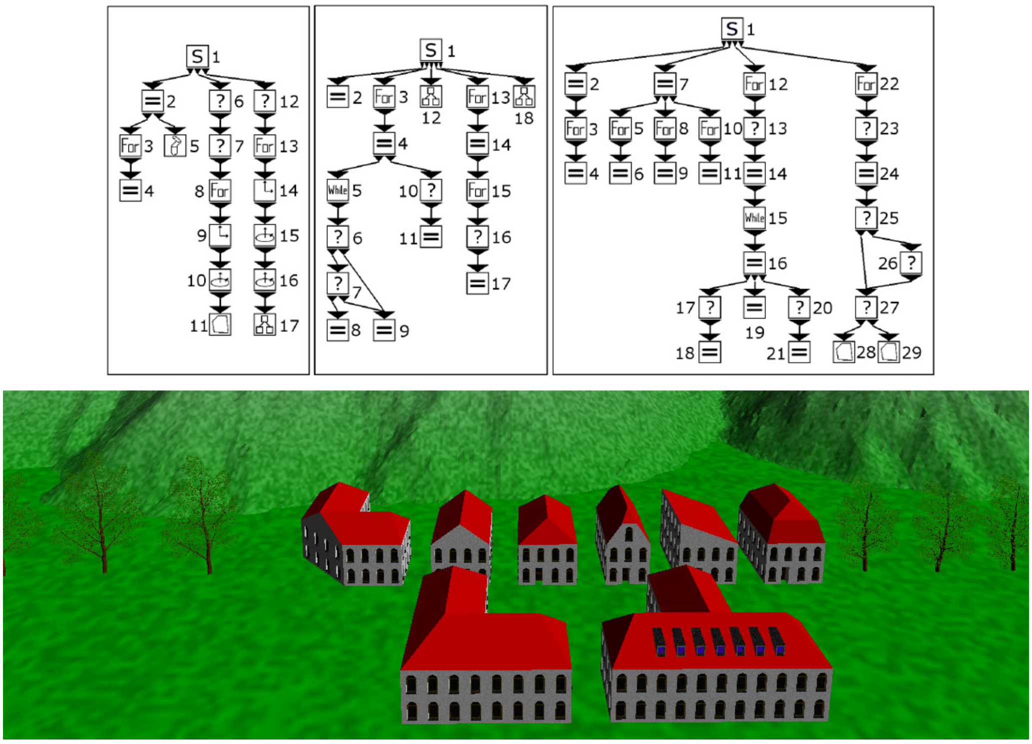


Figure 2.14: *On the top, graphs for tree generation, roof point calculation and roof geometry generation are shown. On the bottom results obtained using such graphs are shown. Image courtesy of Ganster and Klein [BFH05].*

Building interiors Until now we only introduced methods that are mainly targeted at creating the facades of architecture. However, there is also work specifically targeted at interior generation. One method for persistent building interior generation was discussed by Hahn et al. [HBW06]. Their main goal is to enable real-time generation by only creating the immediately needed parts using a lazy generation scheme. Also, persistent direct changes by an artist are supported by storing the change together with a hash value of the position in an external database. The main disadvantage of their method is the restriction of geometry representations to oriented bounding boxes.

2.3 Urban Layouts

Now that we introduced plant and architecture generation, we can look at methods creating whole urban layouts containing those objects. We define urban layouts as the combination of streets, parcels and arbitrary parameter distributions over those elements describing various aspects such as wealth or building types of the city. Using parameters, we can also identify the objects to be placed on parcels, for example arbitrary buildings or plants.

There is substantial work on the procedural generation of urban layouts. Early methods provide artist control mainly by changing global parameters or input data followed by a regeneration of the city. Those methods will be introduced in Section 2.3.1. Later methods also introduce direct and local control to some degree using incremental editing operations. They will be explained in Section 2.3.2.

A general overview of the issues when combining manual editing and procedural generation of virtual worlds is provided by Smelik et al. [STdKB10]. They discuss various manual editing operations an artist could desire and classify them according to their granularity from coarse to fine. Coarse includes rough sketching of the virtual world, while fine deals with modifying single objects. They discuss the open issue of preserving manual changes, but do not provide a new solution to tackle this problem.

2.3.1 Control through global input change

Parish and Müller introduced the first method for procedural generation of urban layouts [PM01]. They introduce a pipeline for city generation, shown in Figure 2.15. A generalization of this pipeline (roads-blocks-allotments-buildings) is the basis for most recent work [VAW$^+$10].

As input, their system uses global parameters such as the main city center or desired street patterns. Local steering is possible using parameter distributions encoded in artist created image maps. Parameters include elevation, population density, zoning (residential, commercial,...), water areas and others. The authors introduce an extended L-system for street generation, able to respond to global goals such as desired road patterns, and local constraints to ensure the generated roads are valid, for example they are not inside a water area. After streets are generated, blocks surrounded by streets are extracted and subdivided into lots. Finally, building geometry is generated using a simple L-system. Artist control of their system is indirect and consists of changing global input parameters or parameter distributions. Small changes in the parameter distributions can have a large effect on the output.

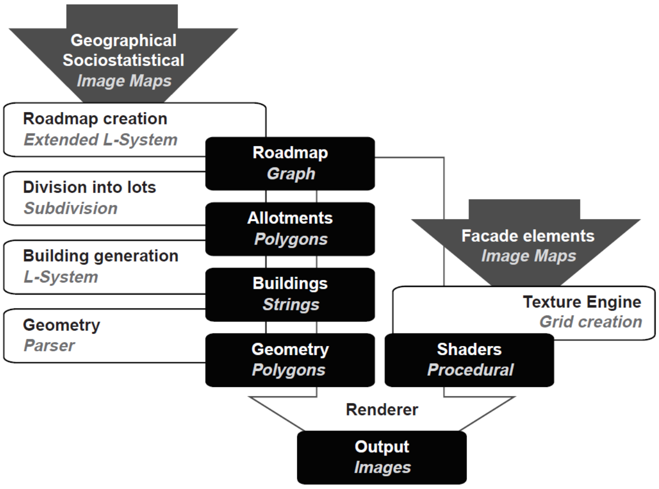


Figure 2.15: *Pipeline for procedural city generation. Image courtesy of Parish and Müller [PM01].*

Direct, low-level editing The street graph network generated by a procedural model can also be edited using traditional interactive editing operations, such as moving vertices (intersections), adding edges (street segments), and deleting edges of the graph. These operations are orthogonal to the actual generation algorithm, and are thus used by most procedural systems. However, the important unanswered problem is how to maintain persistence of such direct edits when global changes are performed. Further, direct edits can result in an invalid layout, for example when streets intersect with parcels. We will provide solutions to those problems in Chapter 4.

Extensions Much subsequent work focuses on parts of the general pipeline described above. Lechner et al. introduce an algorithm for automatic land-use zoning using agent-based simulation [LWWF03]. Glass et. al introduced an algorithm creating street patterns similar to informal, unstructured settlements [GMB06]. An algorithm to simulate virtual humans in procedural cities was proposed by Silveira and Musse [dSM06].

Tensor fields Chen et al. introduced a street modeling technique based on tensor fields [CEW$^+$08]. Their key insight is that hyperstreamlines of a two-dimensional tensor field share a pattern similar to a street network. To exploit this, they introduce design operations to model a tensor field, and show how to extract streets from such a field. To actually model a street network, an artist creates an initial tensor field, representing the input for the street generation. Subsequent refinements of the tensor field reflect indirectly on the generated street network, as shown in Figure 2.16. While direct modifications are *not* supported, this approach provides a more intuitive control of the output compared to L-systems for street generation.

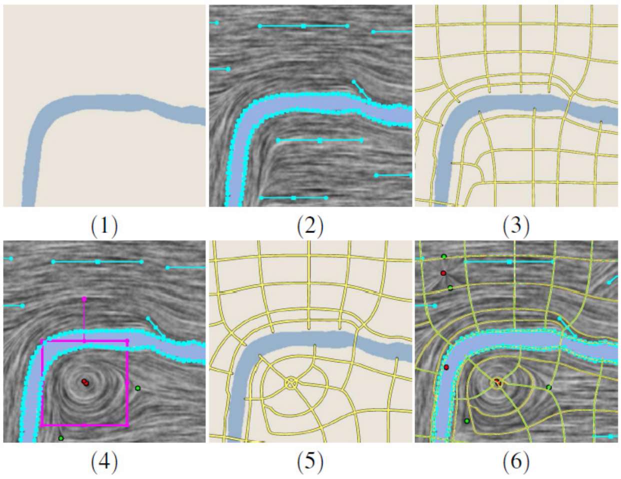


Figure 2.16: *Modeling sequence of streets using tensor fields. (1) Initial landscape (2) Tensor field modeled using intractive tools (3) Streets generated from tensor field (4) Refinements of tensor field (5) New output (6) Further refinements. Image courtesy of Chen et al.[CEW$^+$08].*

Hybrid approach Kelly and McCabe [KM07] proposed a mixture of manual and procedural techniques: Major roads are created manually, while the minor roads enclosed by main roads are created procedurally. This way, the road generation parameters can be distributed in a more fine-grained manner. To help creating the major roads, tools for interactive placement of roads on varying terrain were introduced.

Analysis All the methods presented in this section allow local steering of the generation though locally varying parameters, for example by editing images representing parameter distributions. However, the procedural regeneration after an input change results in (potentially drastic) global changes, even in parts of the model that are distant from the local edit of the parameters. These global changes are difficult to anticipate and control for a designer for a lack of direct control, and prevent any form of direct control mechanisms such as modifying individual streets or lots.

2.3.2 Control through incremental editing

Simulation Based Weber et al. introduced algorithms for interactive city simulation creating a full 3d geometric city model [WMWG09]. Their algorithm works incrementally using time steps. Starting from an initial city, the following steps are performed for each iteration: (1) New streets are planned by stochastically expanding existing streets. Only when a traffic demand model decides that there is enough traffic on the street, it is actually built. (2) When streets enclose a new quarter, a land use simulation is performed for this quarter. (3) Minor streets are generated for new quarters, creating new blocks. (4) New blocks are subdivided to parcels, and buildings are generated. A few possible iterations are shown in Figure 2.17.

After each iteration, the user can change global parameters like the position of the city center or street patterns. Also direct modifications of the street graph are supported.

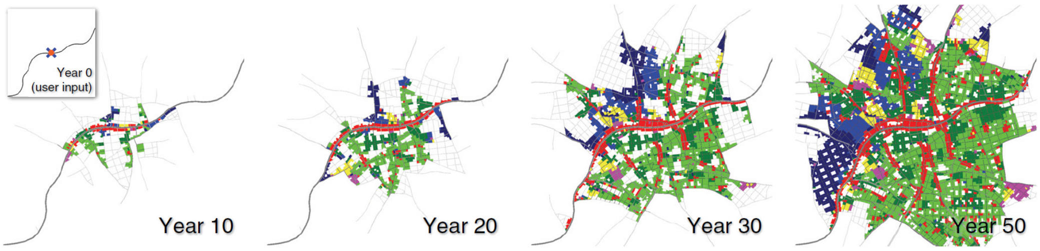


Figure 2.17: *Several iterations of an interactive city simulation over time. Image courtesy of Weber et al. [WMWG09].*

Another simulation-based method was introduced by Vanegas et al. [VABW09]. Their key idea is to tightly couple grid-based simulation of behavioral parameters like population and jobs with geometrical modeling of roads, parcels and buildings. This is done by representing the parameters in a dynamic system, and using an iterative approach: (1) An artist can modify or constrain behavioral parameters using a visual paint-brush style tool. The artist can also directly modify geometrical attributes, for example by creating new highways. (2) After each modification, a dynamic system attempts to create an equilibrium between the behavioral parameters and the geometric aspects of the city. For example, when a new highway is added, the algorithm automatically redistributes the population along the highway. An example is shown in Figure 2.18.

There is also considerable amount of previous work focused on behavioral simulation of cities without geometrical details. We consider this out of scope for this thesis, as geometrical content is needed for many applications in

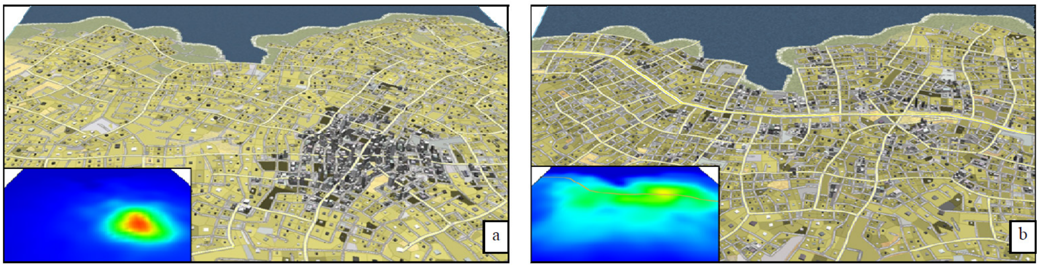


Figure 2.18: *Coupling of behavioral parameters with geometric urban layout: A new highway is added and the population is automatically redistributed. Image courtesy of Vanegas et al. [VABW09].*

computer graphics such as movies or games. Please refer to Vanegas et al. [VAW$^+$10] for an overview.

Example-based method Aliaga et al. introduced an example-based approach for urban layout modeling [AVB08]. Their key idea is to encode the properties of a street network as attributes of street intersection points. These statistical attributes can be extracted from GIS data. A random walk algorithm using those attributes is then employed to connect intersections, creating a random street graph that has the same statistical properties as the example data. Blocks are extracted from this graph and parcels are generated in those blocks using an algorithm based on Voronoi diagrams.

To provide artist control, three synthesis operations are introduced: Join, blend and expand. They allow incrementally combining graphs generated from different example sets. Essentially, they work by placing intersection points based on different examples next to or over an already existing street graph, and the random walk algorithm connects them with new streets. An example output is shown in Figure 2.19.

Further incremental editing operations were introduced by Aliaga et al. in a different paper [ABVA08]. They include moving of a block, cutting and pasting of whole areas and non-linear transformations of whole cities. The main contribution of their work is a constraint solver that minimizes the distortions introduced by an editing operation in the surrounding areas.

Analysis Conceptually, all referenced papers in this section use incremental or iterative editing as basis for artist control. The prerequisite for incremental editing is a procedural generation algorithm able to incorporate a previous output, as visualized in Figure 2.20. Note that simply swapping the output with the input after one iteration is not possible, as input and output are of different type. For instance, in the example-based approach, the input

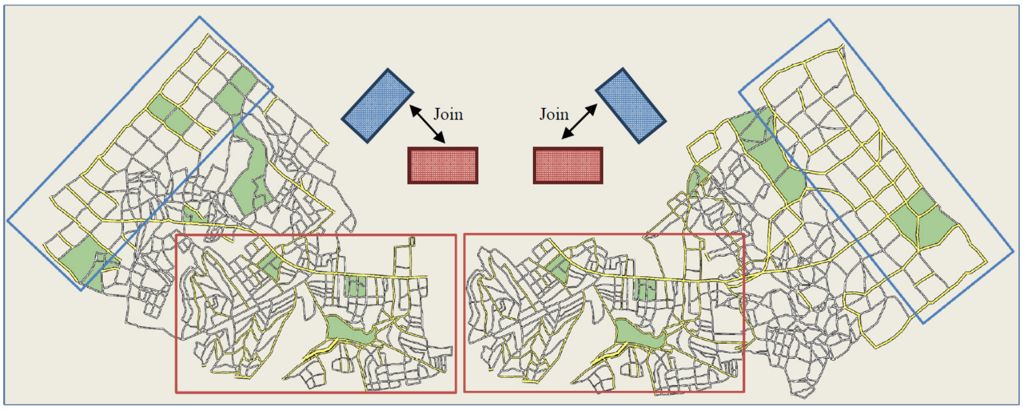


Figure 2.19: *Blue and red areas represent street networks generated from different example data. Using a join and blend operation, they can be connected seamlessly. Image courtesy of Aliaga et al.[AVB08].*

consists of statistical properties of the examples, while the output is an urban layout.

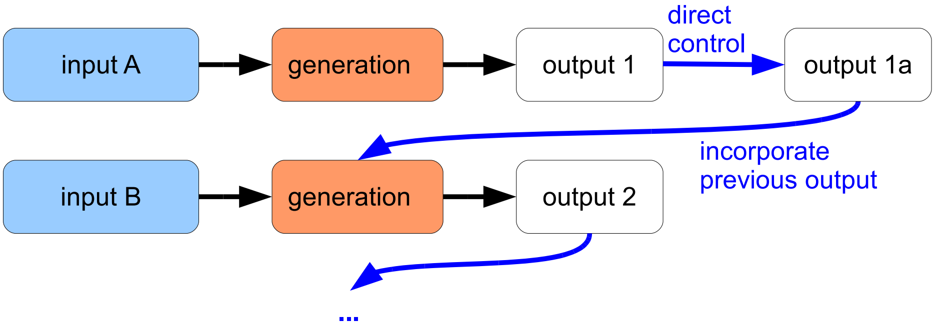


Figure 2.20: *Conceptual view of incremental editing as a basis for artist control.*

The advantages of such methods is that inputs can be changed for each incremental step, and that artists can successively model the output to correspond to their vision. Also direct changes after each iteration are possible. As a major disadvantage, persistence of incremental changes after a global change is not supported. For example, if an artist makes changes to "input A" in Figure 2.20 after "output 2" was created, there is no obvious way to reflect those changes in "output 2".

Further, even though more fine grained control over the generation is possible compared to complete global regeneration, the output of one incremental step can still be difficult to anticipate for an artist, as the involved stochastic processes do not necessarily reflect the intentions of the artist.

2.4 Parallelizing L-Systems

For all the methods introduced thus far it is imperative that they are capable of real-time performance if they are to be used in an interactive modeling environment. Therefore we will now look at performance improvements for some procedural methods. As there are vast amount of procedural methods, we restrict ourselves to L-system based ones. We see this as a first step towards accelerating other methods as well.

The main acceleration idea is to calculate the L-system generation in parallel. In the remainder of this section, we will first introduce methods capable of parallel L-system generation. Then we will introduce basic parallel processing primitives used in our work. Finally, an API for parallel computations on a graphics card will be introduced.

Parallel L-systems Lacz and Hart showed how to use manually written vertex and pixel shaders combined with a render-to-texture loop to compute L-systems [LH04]. Their results (Figure 2.21) show that this is applicable to simple trees. This concept was later extended using automatically generated geometry shaders [Mag09][MK10]. Both methods require a shader compilation step for the productions, and do not support context-sensitive L-systems [MK10]. Further a transformation step of every production's successor to a set of successors is needed to allow independent parallel executions in a shader. For example, the production $L \rightarrow aLf[+L]Lf[-L]L$ is transformed to the set $L \rightarrow aL, af+L, afL, aff-L, aff-L, affL$ [LH04]. This is only valid if the successor of L does not have any effect on the traversal state, which is not generally the case.

Baele and Warzee provides details on how to generate and render seamless branch geometry on the GPU [BW05]. In their approach, the L-system derivation and interpretation is done serially on the CPU, and a shader creates the final branch geometry. We consider this method to be orthogonal to L-system parallelization, as it does not parallelize the derivation and interpretation. An example is shown in Figure 2.22.

An algorithm utilizing multiple processors (the results show up to 8 CPUs) with distributed memory, communicating using the Message Passing Interface (MPI) was introduced [YHL+07]. In their algorithm, the derivation of the L-system is performed using two binary trees, a Growth-State Tree (GST) and a Growth-Manner Tree (GMT). To actually render the system, the GST is interpreted as a scene graph. In order to get global scene-graph transformation matrices needed for rendering in the individual threads, the matrices are serially transfered from one process to the next. As the memory

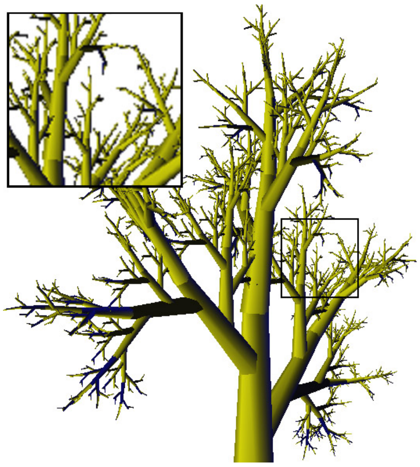


Figure 2.21: *This tree was generated on the GPU. Image courtesy of Lacz and Hart [LH04].*

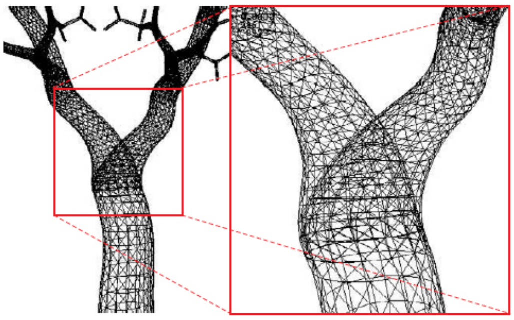


Figure 2.22: *Seamless branches generated on the GPU. Image courtesy of Baele and Warzee [BW05].*

is distributed between processors, special care is taken to transfer memory between processors in order to enable context searching. In CUDA as well as on multi-core CPUs we have a shared global memory, eliminating the need for memory transfers in context-sensitive systems.

Parallel primitives In Section 5 we will extensively use the parallel scan primitive in our algorithms. This primitive is the basis for most parallel algorithms. Given an ordered set of values $[a_0, a_1, \ldots, a_n]$ and an associative operator $\circ$ with the identity element I, an exclusive scan operation will result in the ordered set $[I, a_0, a_0 \circ a_1, \ldots, a_0 \circ a_1 \circ \ldots \circ a_{n-1}]$. A very efficient parallel algorithm based on hierarchies exists for those operations [SHZO07]. If the operator is the addition, this results in a set of values s_i with $s_i = \sum_{j=0}^{i-1} a_j$. An example is shown in Figure 2.23.

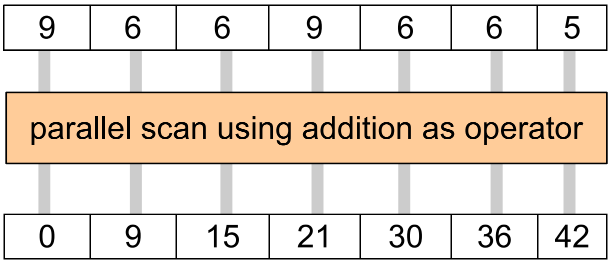


Figure 2.23: *Parallel scan using addition.*

The main advantage of the scan primitive is its capability to compute seemingly serial operations very efficiently on highly parallel hardware, since subsequences can be processed independently due to associativity. Unless noted otherwise, we always refer to an exclusive scan on integral values using the addition operator when we use the term scan in this thesis.

Parallel computation in CUDA In order to access the parallel computing capabilities of GPUs, we employ the NVIDIA CUDA data-parallel programming framework [COR07]. CUDA abstracts the cores with SIMD functionality of a GPU into a hierarchy of threads and blocks, as shown in Figure 2.24.

Recent work shows how to map computations having a highly dynamic nature to CUDA. Most notably, algorithms to efficiently implement work-load balancing using a compactation step were introduced in the context of KD-trees [ZHWG08], Reyes-style subdivision [PO08] and bounding volume hierarchy construction [LGS$^+$09]. Generalized stream compaction was presented by Billeter et al. [BOA09]. In the context of tessellating parametric surfaces, scan operations were used in order to scatter dynamically generated vertices to a VBO [SS09]. We employ both work-load balancing and vertex scattering in our work.

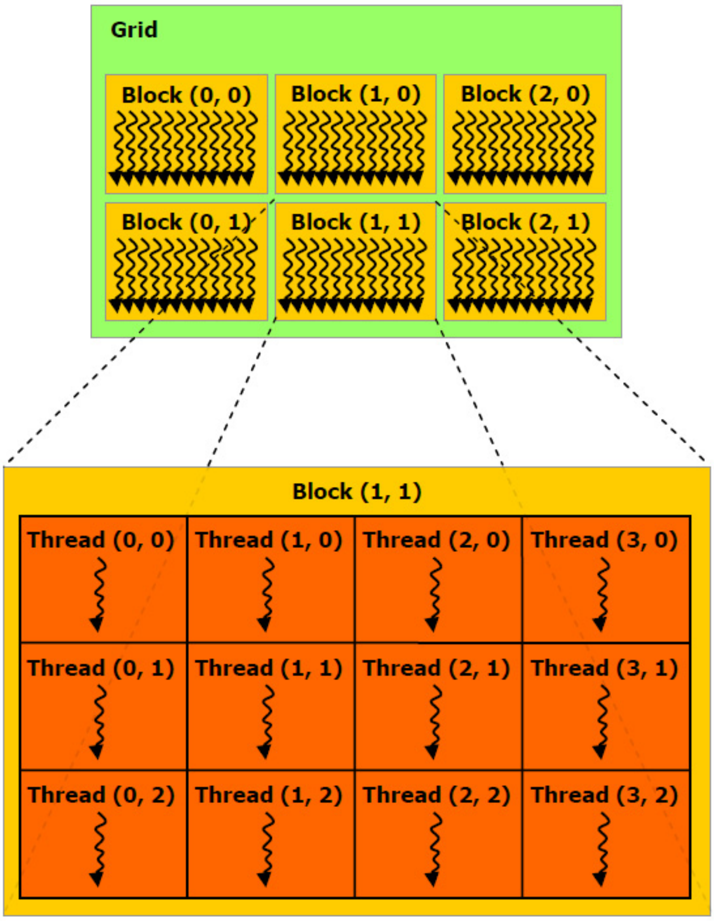


Figure 2.24: *CUDA hierarchy: multiple threads are organized into blocks. Each GPU core can run one block a time, while multiple blocks are scheduled automatically on different cores. Image courtesy of NVidia [COR07].*

3

Direct Artist Control for Procedural Architecture

Figure 3.1: *Screenshots from our real-time editor for grammar-based procedural architecture. Left: Visual editing of grammar rules. Middle left: Direct dragging of the red ground-plan vertex and modifying the height with a slider creates the building on the middle right. While dragging, the building is updated instantly. Right: Editing is possible at multiple levels, here the high-level shell of a building is modified.*

In the thesis introduction (Section 1), we mentioned the problems of persistent direct control, the importance of a visual interface for input definition, as well as real-time performance. In this chapter we will provide solutions for those problems specifically targeted at building generation. In Figure 3.1 a few examples of the resulting system are shown. The results of this chapter were published under the following reference: [LWW08].

Let us first explain the mentioned problems in the context of building generation: The current approach to 3D modeling of buildings is to manually create 3D geometry using tools like Autodesk Maya or 3ds Max ™. This process is time consuming, tedious and repetitive, but allows the artist full control over every aspect of the final 3D model. Recently, grammar-based procedural modeling has shown promising results, for example for architectural modeling [WWSR03, MWH+06]. However, these approaches allow only *indirect* control over the final model by changing the underlying grammar, or *global* control by changing some parameters. When more fine-grained control over individual buildings is needed, a tedious change grammar-regenerate- ... cycle is required until the desired output is achieved. Furthermore, current procedural modeling systems are mostly text based and therefore impractical for the intended users, i.e., artists and technical artists.

In this chapter, we aim to take the next step in procedural modeling by combining the full generative power of design grammars with the ease of use and flexibility of 3D modeling systems. This is achieved by introducing visual editing, with direct local control of all aspects of the grammar.

Direct Visual Editing While text-based production systems are very powerful, end users require a visual frontend to be able to use them productively. It is important to provide both a visual rule editor to create and modify the individual rules of the grammar and immediately see the consequences for the generated models, as well as a visual model editor which allows modifying the instances generated by the rule derivation process.

Local Modifications Assume the artist wants to assign a different texture, different window width or different ornamentation rule to a specific window on a facade. In a current text-based procedural modeling system, the artist would have to write several new rules to identify the floor and column of the window and add the modification. In a visual editor, the desired workflow is that the artist simply *selects* the desired window and chooses a new texture, rule or window width. To make this happen, we need to solve the problem how to allow *local* modifications to variables, rule selection and geometry, *without* having to change the underlying grammar.

Semantic and Geometric Selection Local modification should often be applied to several elements, not just one. For example, all windows in a specific floor or column should have a changed appearance. We therefore need to provide mechanisms to select elements based on *semantic* attributes like facade number, floor or column, which are not limited by the derivation hierarchy. These should be paired with standard tools known from 3D modeling like selection rectangles etc.

Persistence Local modifications are in a sense more volatile than actual grammar rules. Assume, for example, that a specific window on a facade of a building is modified. If the next modification is to change the height of the building, the whole instance has to be regenerated from the grammar rules. In order not to lose the previous modification, any modification has to be stored *persistently* – this is especially difficult when the structure of the grammar changes.

Main Contribution The main contribution of this chapter is to enable a visual editing workflow for grammar-based modeling by providing the aforementioned functionalities. In particular, we introduce a set of visual operators for both rule and building editing, and introduce the concept of *exact* and *semantic locators*, which will be used to allow local modifications and semantic selections, as well as to solve the persistence problem. Local modifications are a true extension of the expressive power of design grammars, and bring procedural modeling a step closer to a workflow acceptable to artists.

The remainder of the chapter is structured as follows: At first we provide a description of our visual editing concepts in Section 3.1. Section 3.2 gives details and discusses how we use exact and semantic locators, while Section 3.3 discusses the visual building and rule editors. In Section 3.4 the workflow and performance is evaluated, and implementation details are provided. In Section 3.5 a discussion is provided comparing our method to previous work.

3.1 Visual Editing Concepts

In this section we provide an overview of design grammars and the problems that occur during visual editing, and our concepts to solve them.

The main concept of a design grammar as used for example for architecture is based on a *shape grammar* utilizing a rulebase, as already introduced in Section 2.2. In this chapter we loosely follow the notation of CGA shape as introduced by Müller et al. [MWH$^+$06], while the concepts work for other grammars as well. We will now briefly repeat the most important concepts of CGA shape:

Starting from an initial axiom shape (for example a ground plan), rules are applied, replacing shapes with other shapes. A rule has a *labeled shape* on the left hand side, called *predecessor*, and one or multiple shapes and *commands* on the right hand side, called *successor*. Commands are macros creating new shapes or commands. Three commands were introduced in [MWH$^+$06]: *Split* of the current shape into multiple shapes, *repeat* of one shape multiple times and *component split* creating new shapes on components (e.g. faces or edges) of the current shape. Every rule application creates a new *configuration* C_i of shapes. During a rule application, a hierarchy of shapes is generated corresponding to a particular *instance* created by the grammar, by inserting the successor shapes as children of the predecessor shape. This *production process* is executed until only terminal shapes are left. An example rulebase is visualized in Figure 3.2, from this rulebase the instance and associated shape hierarchy in Figure 3.3 are *automatically* generated.

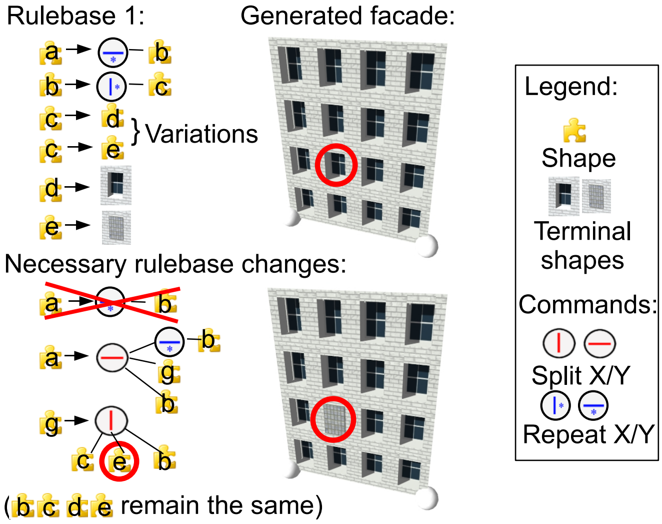


Figure 3.2: *The rulebase on the top has two possible windows, enabling variations during generation. An example output is shown on the top right. If we want to specify the window type to be used for the encircled window, we have to manually rewrite rules in order to set the window. The necessary rulebase changes are shown at the bottom, creating the new rendering. We found this rewriting to be tedious and error prone, even when just one variation is controlled.*

The power of design grammars lies in their capability to produce *variations*. This means that each instance created by the grammar will look different. The following mechanisms are available to introduce variation in design grammars:

1. multiple possible production rules for a shape (chosen stochastically)
2. parameters (e.g., window width) chosen according to variables set by the user (in a text file)
3. random parameter assignments in rules

Note that all these mechanisms are *global* in nature, i.e., they are typically only chosen once for a whole instance. If a shape that uses a particular variable appears in several nodes in the shape hierarchy, there is no way to assign different values to the different nodes. Instead, for each node that should differ from the rest, a set of new rules have to be introduced which expose the desired variability via new variables, as shown in Figure 3.2. This is tedious and quickly leads to an explosion of the rulebase.

Direct Control of Variation In this chapter, we propose a new paradigm for rule-based modeling: *direct local* control of the shape hierarchy, shown in Figure 3.3.

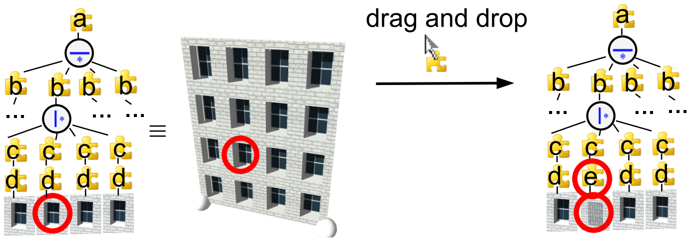


Figure 3.3: *On the left, the automatically generated shape hierarchy corresponding to the facade in Figure 3.2 is shown. Only the second floor is visualized, in order to increase readability. Utilizing direct control, the user can drag and drop the desired window on the rendered facade, automatically changing the underlying shape hierarchy, as seen on the left. No manual rewriting is necessary for the user.*

At first, we provide tools to directly modify the shape hierarchy of an instance via so-called *locators*, which are stored externally to the grammar, thus decoupling local modifications from the grammar. Direct changes are important to create an artist-controlled unique look for a specific instance. In particular, we introduce the following operators which can act locally on any level of the shape hierarchy:

1. modify the variables used in a particular node in the shape hierarchy
2. select the production rule applied to a shape (selected from multiple possibilities given by the grammar, or even arbitrary rules)
3. pin random choices
4. directly set the geometry and textures used in a particular terminal shape

Note that the items selected in 2 to 4 can be (automatically) expressed as variables and are therefore special cases of 1. While these operators may seem to be straightforward to implement, actually they are not. Since these direct modifications need to be provided in a *visual editor*, we have identified two major problems:

Selection Problem Every variable is used in at least one, but typically in several nodes in the shape hierarchy. For example, the window width is a variable that can be set once for a building and is used in every window tile. Obvious ways to influence the window width are to set one width for all windows (the default), or to set the width of some windows individually. However, the most common required action will be to change the variable for a certain subset of nodes in which the variable is used. One solution is to provide *hierarchical* assignments, i.e., a variable can be assigned a value at any node in the hierarchy. However, that is often not sufficient. We also require *semantic* assignments, for example, selection of all windows in a floor, or all windows in a column. Both subset selection methods are shown in Figure 3.4.

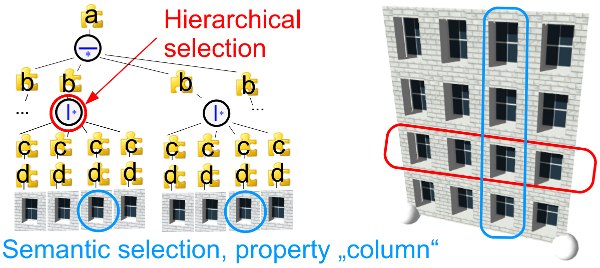


Figure 3.4: *Using hierarchical selections, all shapes underlying a specific shape in the shape hierarchy are selected. Semantic selections allow selecting multiple shapes that share common semantic properties. Please note that it is impossible to select a whole column just by using one hierarchical selection in this shape hierarchy, as there is no rule that directly represents a whole column.*

Persistence Problem Most modifications $m_i, 1 \leq i \leq 4$, require the instance along with its shape hierarchy to be regenerated from the grammar rules. However, as shown in Figure 3.5, this will obliterate any previous modification to the instance. Therefore, any modification needs to be stored persistently. However, rebuilding the shape hierarchy could lead to a different hierarchy, so an important problem to solve is how to apply a modification to a different shape hierarchy.

We solve both of these problems using *semantic annotations* which can easily be specified in the visual editor, and provide some heuristics how to set these annotations automatically. We define two types of instance locators: *exact instance locators* and *semantic instance locators*. An exact locator stores the exact location of a shape in the shape hierarchy. This is essentially done by storing all information occurring along the path from the shape to the axiom. Using an exact instance locator we can *unambiguously* locate

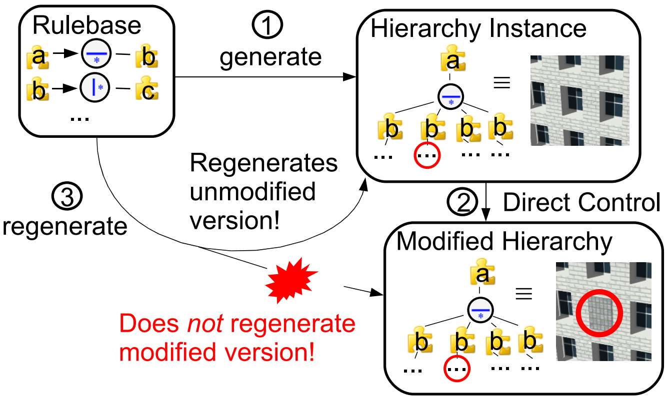


Figure 3.5: *A persistence problem occurs when we at first generate a shape hierarchy from the rulebase, and then modify this hierarchy utilizing direct control. When we need to perform a regeneration (for example because the house height has changed) the unmodified version is generated, thus all direct modifications are lost.*

shapes, and thus retain selections after a regeneration. However, this does not allow semantic selections based for example on floor and column numbers. We therefore introduce *semantic tags* that can be attached to structural commands in the shape hierarchy (e.g., split and repeat commands).

Visual Rule Editing Finally, we provide an intuitive user interface for editing the grammar in a visual editor. We have identified the following important operations, which will be further explored in Section 3.3:

- graphical assignment of numeric values (i.e., dragging split planes)
- multiple views on a rule: rendering, visualization of commands and structural overview
- focus and context visualization of rules

3.2 Instance Locators for Local Control

Three problems emerge when we want to enable local control. At first, we have to describe how we *select* shapes in the hierarchy. Secondly, we must define how to *apply modifications* to those selections. Thirdly, those modifications have to be *persistent*. We will describe solutions to those problems in this section, which rely mainly on the concept of *instance locators*.

3.2.1 Selections

There are three types of selections we want to provide: Selection of a *single* shape in the hierarchy, *hierarchical selections* and *semantic selections.* Essentially a selection works as follows: (1) The user clicks on the 3D rendering. Using intersection tests, we calculate which shape in the shape hierarchy the user has clicked on. (2) For this shape an *exact instance locator* or a *semantic instance locator* is calculated and stored in *loc*. (3) To actually highlight all selections in the rendering, we use the locator *loc* by comparing it to the (temporarily created) locator of each shape in the hierarchy, and highlight all matches. Let us now define instance locators in more detail:

Exact Instance Locator The selection of a *single* shape in the shape hierarchy is specified the following way: For every shape or command in the hierarchy graph, we sequentially number every outgoing edge from left to right starting from 1. An example numbering is shown in Figure 3.6. In order to specify one shape s, the unique path p from the axiom to s is determined. We walk along this path, and write every shape, command and edge number we encounter into an ordered tuple. We call this tuple *exact instance locator*, as it allows us to uniquely specify a single shape instance. The resulting exact instance locators for Figure 3.6 are shown in Table 3.1. Note that it would theoretically suffice to use edge numbers only, but we also include shapes in the locator to be able to check whether the locator is valid after a hierarchy change.

A hierarchical selection is simply a selection of a single shape that is an internal node of the shape hierarchy, and therefore can also be expressed with an exact instance locator.

Semantic Instance Locator Using just the shape hierarchy and exact locators, there are two problems:

1. We cannot select shapes based on semantic attributes, for example *2nd column on the 3rd row.*
2. We cannot perform semantic queries in the form *select all shapes in the 2nd column.*

Those problems are caused by the lack of semantic information in the shape hierarchy. We therefore introduce *semantic tags* that can be attached to split or repeat commands. For example, in Figure 3.7 the semantic tags $floor$ and $column$ are attached to rules. The tag $facade$ can be used to differentiate facades. Actually defining semantic tags is very easy for the user:

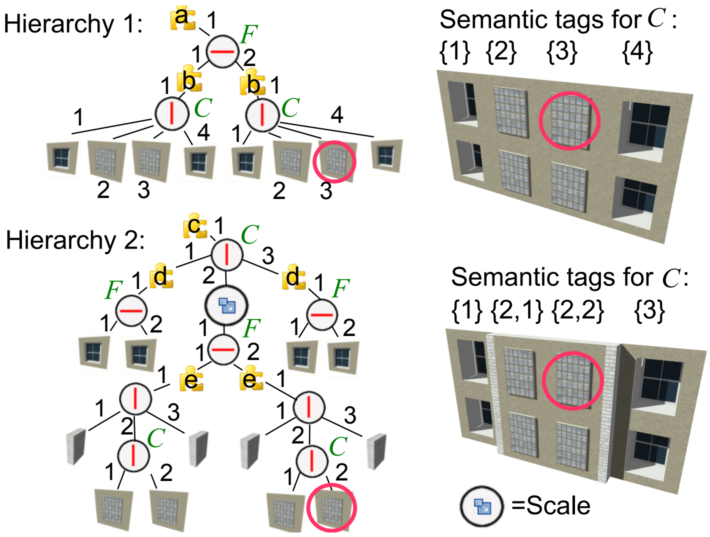


Figure 3.6: *Hierarchy 1 corresponds to the rulebase in Figure 3.7, hierarchy 2 is a more complex example. Red circles around shapes represent corresponding selections. Edges are sequentially numbered. Over the bottom rendering, the semantic tags and corresponding absolute values of columns are shown. Please note that while the renderings are quite similar, the underlying graphs are significantly different.*

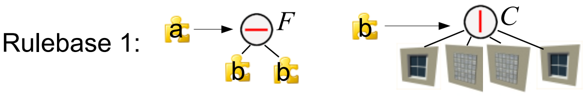


Figure 3.7: *We introduce semantic tags attachable to commands, represented here as F for floor and C for column.*

In our rule editor, a simple drag and drop operation of a tag on a command applies the tag. This has to be performed only once when defining rules, everything else is done automatically. Please note that in Figure 3.6 (bottom) not every vertical (X)-split has the tag *column* applied, in this way the user can specify what is considered a unique column and what not.

Automated semantic tagging is possible by using the following heuristic: The user specifies threshold f_m for the height of multiple floors and f_s for the height of one floor. Y-Splits/Repeats where the scope height is above f_m are automatically tagged as $floor$, directly succeeding shapes or commands having a scope height below f_s are excluded from this tagging. Analogously X-Splits/Repeats are tagged as *column* using the width as threshold. All component splits having a height above a threshold are tagged as $facade$.

A *semantic instance locator* for shape s can now be constructed from a hierarchy graph the following way: At first, we construct the unique path

p from the axiom to s. We walk along this path, and when we encounter a tagged command, the assignment $tag = edgenumber$ is added to the locator. $edgenumber$ is the sequential numbering of the edge to the next shape already introduced for exact instance locators. It is important to note that tags can occur multiple times in the locator. By using the sequential edge number, we exploit the spatial ordering split and repeat rules provide. Example semantic instance locators for the encircled shapes in Figure 3.6 are shown in Table 3.1. We can improve the granularity of semantic locators by additionally saving the symbol of shape s.

Marked S.	Exact Instance Locator
Hierarchy 1	$\{a, 1, S_y, 2, b, 1, S_x, 3\}$
Hierarchy 2	$\{c, 1, S_x, 2, Scale, 1, S_y, 2, e, 1, S_x, 2, S_x, 2\}$

Marked S.	Semantic Instance Locator
Hierarchy 1	$(F = 2, C = 3)$
Hierarchy 2	$(C = 2, F = 2, C = 2)$

Table 3.1: *Exact and semantic instance locators for the encircled shapes in Figure 3.6 are shown here.* S_x, S_y *represent a Split X/Y command, Scale a scale command.*

Using semantic instance locators it is easy to solve the previously mentioned problems:

1. We can perform selections based on semantic attributes simply by specifying a semantic locator, and searching for shapes with matching locators.
2. Selections of whole columns are done by ignoring the $floor$ tags during selection searching. For entire floor selection this works analogously.

Semantic assignments are assignments applied to shapes having a specific semantic tag. Please note that semantic assignments can be flexibly combined with hierarchical assignments, by attaching semantic assignments to internal nodes of the hierarchy graph.

Exact instance locators are used by the rule editor to specify selections in the GUI. Both exact and semantic locators can be used in the building editor.

Anchor Points We can construct locators relative to an *anchor point*. Possible Anchor points are either left, right, or center in horizontal direction and bottom, top, or center in vertical direction. Per default the closest anchor point is chosen (with priority on the borders for equal distance to the center), because that proved to be most intuitive in our experience. Additionally, the user has the option to modify an anchor point. The actual generation of anchored locators is straightforward: Instead of $tag = edgenumber$ we use $tag = numEdges - edgenumber$ for locators anchored to right or top, and $tag = numEdges/2 - edgenumber$ for locators anchored to the center. For example, using anchored locators it is very easy to specify a selection that should always contain the center column.

3.2.2 Direct Modifications and Persistence

Using a selection specified in an instance locator, we can specify *where* a modification should be performed. The next step is to define *what* should actually be modified. As we already mentioned in Section 3.1, all local modifications (node variables, rule to use, random choices, geometry and textures) can be expressed as variable assignments, thus a variable assignment describes *what* to modify.

Using both a locator loc_m and a variable assignment v_m, a modification is exactly specified. In order to allow hierarchical assignments, all variable assignments have the following properties: The assignment extends its scope to the underlying *subtree*, thereby modifying the value of variables used in any rule application of this subtree. Subsequent assignment on lower levels override assignments on higher levels, thus providing local control for every shape. An example assignment is shown in Figure 3.8.

The key to *persistence* is that instead of applying v_m to the current hierarchy, we *save* loc_m *and* v_m *externally.* Of course we can save multiple modifications externally, thus preserving previous modifications.

The actual application of v_m is carried out in the following way: First we delete the current shape hierarchy, and start the production process from the axiom (configuration C_0). Now, every time we insert a shape s into a configuration C_n, we create an instance locator loc_s for s. When $loc_m = loc_s$, we attach the variable assignment v_m to s. Note that in a longer editing session, some variables can be assigned different values using the same locator. In this case later assignments override previous ones.

Transformations between Hierarchies Semantic locators are normally quite robust with respect to modifications in the visual editor, like changes of the floor plan, modifications of variables etc. However, sometimes the hierarchy changes in a way that a semantic locator does not fit the hierarchy anymore. This can happen when the rulebase is edited in the rule editor, or when substantial changes are made to the choices of productions in higher levels in the shape hierarchy. Still, we want to give the user the possibility to

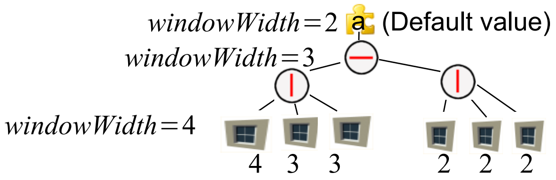


Figure 3.8: *On every shape we can assign values to variables. Assignments extend their scope to all underlying shapes. Assignments on lower levels override assignments on higher levels. Numbers below windows show the values of* $windowWidth$.

recover modifications and apply them to the new hierarchy by *transforming* the locator. Please note that such an algorithm can also be used to transfer modifications from one building to another.

We classify semantic locators into three categories: (a) valid, (b) structurally invalid and (c) semantically invalid. Valid locators correspond to a path in the current shape hierarchy. A locator loc_m is structurally invalid when a *prefix* of loc_m matches the locator loc_s of some shape s in the shape hierarchy, but the *edgenumber* following the prefix is greater than the number of outgoing edges of s. This usually happens when resizing scopes, causing subtrees attached to repeat commands to disappear. Finally, semantically invalid locators are those that are not valid or structurally invalid, i.e., they do not fit the current shape hierarchy at all.

In general, artists prefer direct control over heuristics that run automatically. Therefore, in order to deal with invalid locators, we offer the user a command to transform semantically invalid locators to the current hierarchy. In our implementation a list of locators is offered, color coded according to their category. By selecting a semantically invalid locator from this list and confirming with a button, this locator can be automatically transformed to the new hierarchy using the algorithm described in the following. Note that it does not make sense to transform structurally invalid locators since they often become valid again due to further operations (resizing, ...).

In the following we describe how to transform a semantically invalid locator using so-called *sequential orderings* for semantic tags. Optionally, all calculations can be restricted to the subtree of the shape hierarchy that does not include a possibly matching prefix in order to preserve the maximum semantic context. The idea is to assign one *ordering value* per tag to every shape, regardless of how often the tag appears in the semantic locator. The ordering value depends on the spatial directions of the tags encountered along the way, since we typically have tags “split” by other tags. When externally saving semantic locators, we also save the corresponding ordering values.

Example sequential ordering directions are shown in Figure 3.9 (top). Using this sequential ordering, a transformation of a semantic locator *loc* is done the following way: We compare the *ordering values* (instead of the edge numbers) of this locator with the ordering values of the other hierarchy. When a match is found at shape s, the transformed semantic locator is loc_s.

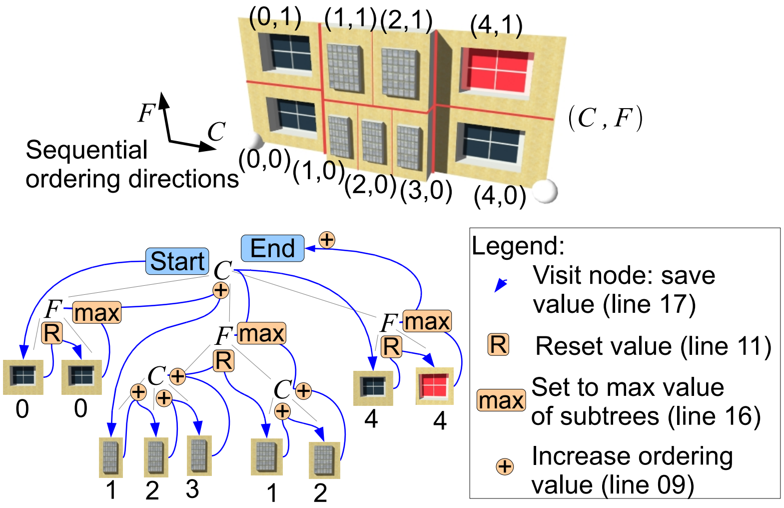


Figure 3.9: *Example for ordering value calculation. The rendering corresponds to the hierarchy graph. To increase readability, we only show elements having a tag attached in the graph, as other elements do not influence the algorithm. Using a blue line we illustrate the traversal of the algorithm during calculation of tag* C*, while orange circles highlight important events occurring during the traversal. The resulting sequential numbering is overlaid in the rendering. Line numbers correspond to Figure 3.10.*

The algorithm to calculate ordering values for a tag `currentTag` and each node in the shape hierarchy is illustrated as pseudo code in Figure 3.10 and works as follows: A modified post-order traversal of all nodes is performed, and after each visited subtree it is decided if the ordering value `seqValue` should be increased. Basically a node `node` associated with `currentTag` increases `seqValue` by one if `currentTag` does not occur in the last visited subtree. There are two possibilities if a different tag `otherTag` is associated with `node`: (1) That tag is ignored for the calculation of `seqValue`. (2) The tag is defined to be a "semantic splitter", i.e., `seqValue` is calculated in each of the subtrees of the tag *independently*. The overall increase of `seqValue` is determined by the *maximum* increase in any of the subtrees. For example, floors and columns are mutual semantic splitters, thus preventing a column tag to be counted on multiple floors. Semantic splitters for arbitrary tags can be defined in a table, but usually correspond to tags associated to splits/repeats in alternating coordinate axes.

```
currentTag = ...; ordValue=0; //Init values
01:CalcAbs(node) {
02: isSplitter = Is tag attached to node semantic
                 splitter of currentTag?
02: hasCurrentTag = Is tag attached to node equal currentTag?
03: previousOrd= ordValue;  //Save current ordering value
04: int maximum= 0;
05: for all children of node {
06:  Postorder traversal: recurse into CalcAbs(child)
07:  //After subtree was visited:
08:  if (hasCurrentTag&(currentTag not found in subtree ))
09:   ordValue++;  //Increase ordering count!
10:  if (isSplitter) { //Special handling of splitters
11:   ordValue= previousOrd; //Restore previous ordValue
12:   maximum= max(ordValue, maximum);
13:  }
14: }
15: if (isSplitter)
16:  ordValue= maximum; //Set to maximum subtree count
17: node.tag.ordValue= ordValue; //VISIT node: save value
18:}
```

Figure 3.10: *Pseudocode for ordering value calculation. Essentially this is a modified postorder traversal with special measures to increase the count of the ordering value after each subtree was visited. Lines that handle arbitrary nesting of tags are marked orange, lines that actually increase the ordering value are yellow.*

Please note that this algorithm exploits structural information created by split/repeat commands, *not* actual world-space positions of shapes. Therefore, when scope translation commands occur in lower-level shapes, it may produce incorrect sequential orderings—in the worst case this can lead to modifications being transferred to shifted positions. To prevent this, world-space positions would have to be considered for shapes with translation commands, for example by casting a ray along the ordering direction to determine the world-space ordering.

3.3 Interactive Visual Editor for Grammars

We will now explore how the introduced concepts fit together creating a new visual editing paradigm for grammars. Inspired by the observation of direct versus indirect control, we have separate windows for a building editor and a rule editor in our GUI, seen in Figure 3.11. The rule editor provides indirect control, allowing the creation of rulebases from scratch, and the building editor provides direct variation control to the artist. All windows can run concurrently, and are linked in several ways: It is possible to edit a rule, and show the effects of this edit on a building. Further drag and drop functionality is provided between the windows, allowing for example direct application of textures.

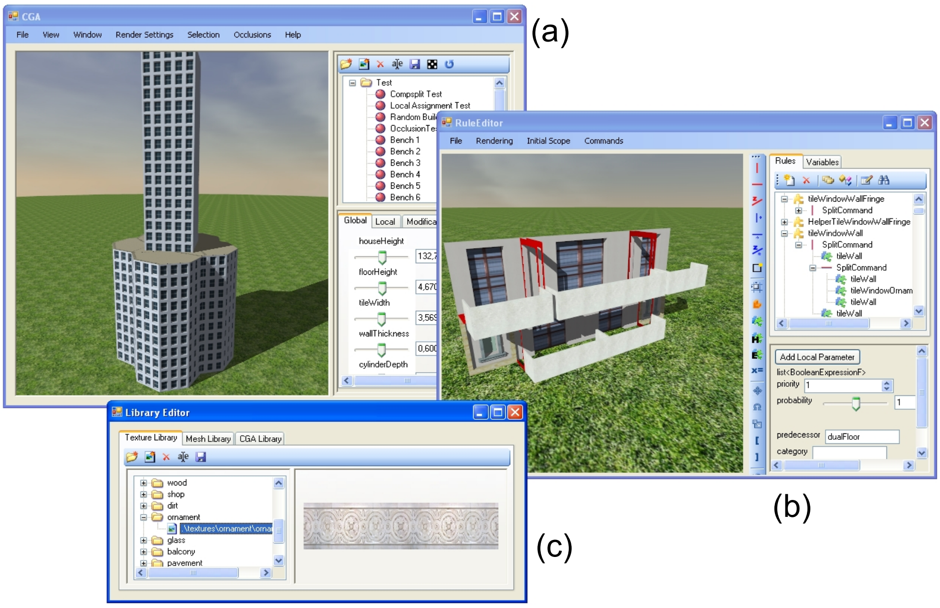
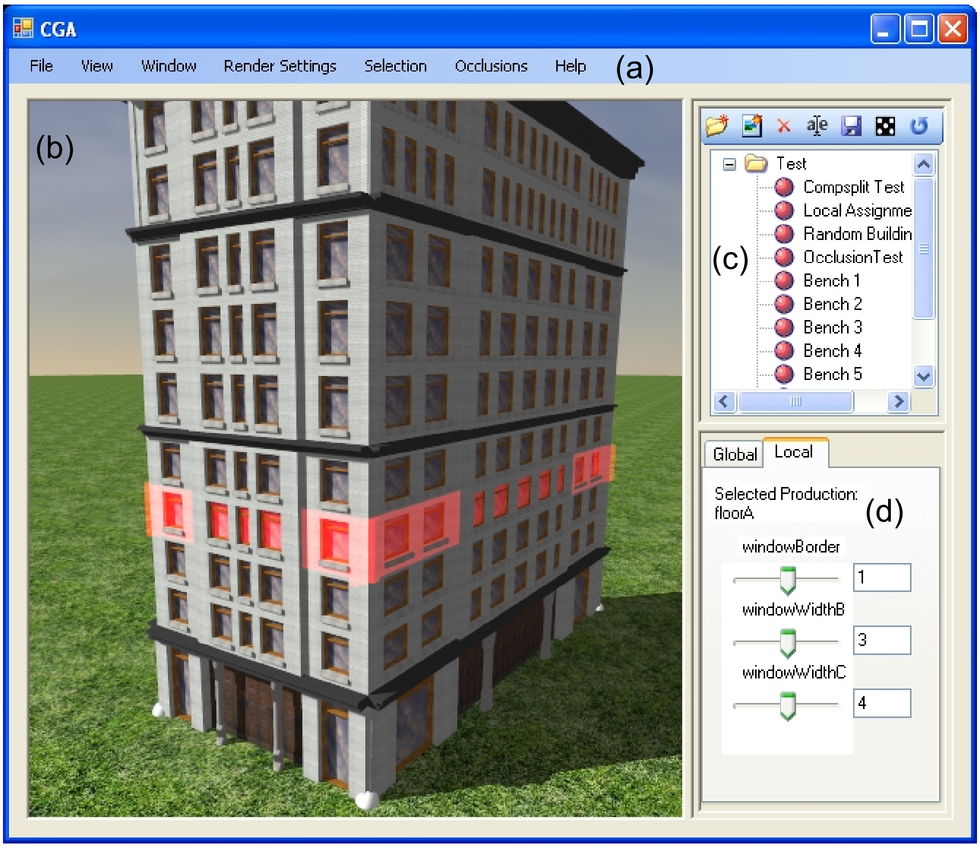


Figure 3.11: *Three windows make up our GUI: (a) A building editor enables direct variation control on buildings (b) Rulebases can be visually created from scratch in the rule editor, providing indirect control (c) Textures and meshes are stored in the library editor.*

3.3.1 Building Editor

In our building editor shown in Figure 3.12, we want to allow direct variation control to configurations, without changing the rulebase. In order to specify the position of a variation, we need to enable selections of shapes at first. This is done using *picking* in the 3D rendering: A user can click on a shape, we internally create an *instance locator* for this shape. In order to specify *hierarchical selections* we provide a button to move the selection up one level in the hierarchy. *Semantic selections* can be specified by using checkboxes for "On Whole Row", "On Whole Column" or "On All Facades" in the menu.

To actually perform modifications on selections, we expose controls for variable aspects: Sliders are automatically created for every parameter occurring in the current selection. The user can now locally adjust those parameters, implicitly creating a *variable assignment* attached to the current selection. In order to specify a production rule to be used, the user can *drag and drop* a specific production from a rule library on the selection. Modifications of textures and meshes are performed by *drag and drop* from a texture or mesh library. Persistence is maintained as previously described: We store the instance locator together with the modification to be performed externally.



Selected Production: floorA

windowBorder 1

windowWidthB 3

windowWidthC 4

Figure 3.12: *Interactive building editor providing direct variation control. (a) Menu with various rendering and derivation controls (b) Real-Time rendering of result. Currently a floor is selected. (c) List of all building instances (d) All parameters occurring in the currently selected shape.*

An example showing hierarchical selections and modifications is shown in Figure 3.13. Hierarchical selections allow artists to specify the granularity of their direct modifications.

3.3.2 Rule Editor

We implemented a rule editor based on the language elements of CGA shape [MWH$^+$06], all of which are visually editable, and we can create rulebases from scratch. Three views on the currently edited rule make this possible:

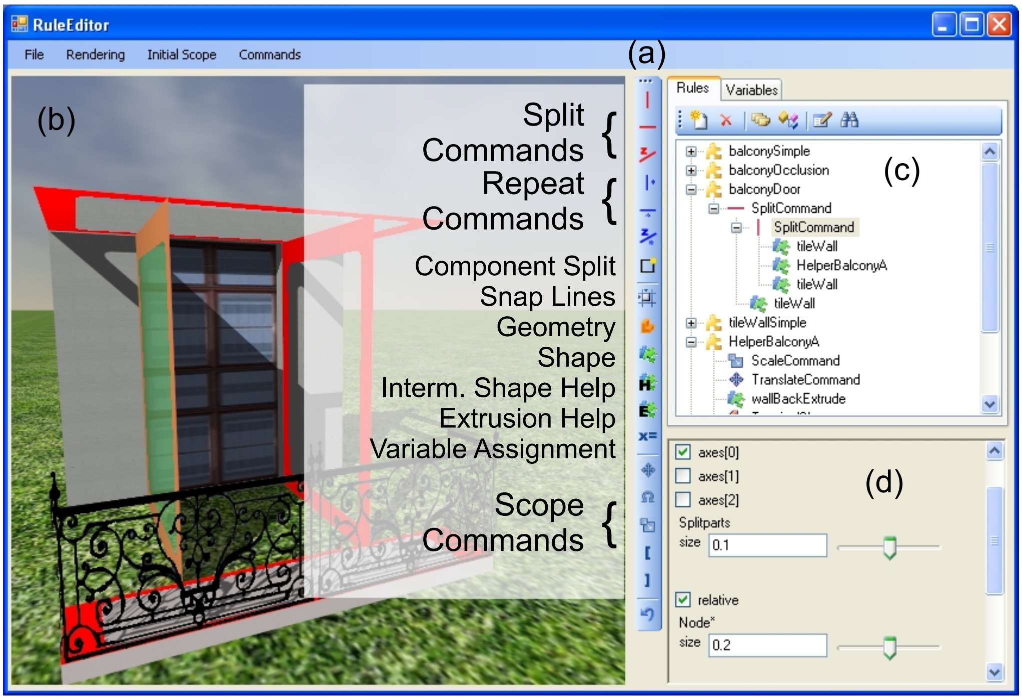
First, a 3D rendering of the derivation overlaid with a visualization of shapes and commands provides direct visual feedback, seen in Figure 3.14(b). Second, a treeview displaying all rules occurring in the current derivation allows easy navigation and provides an overview, as seen in Figure 3.14(c).

Figure 3.13: *Sequence showing hierarchical modifications. (1) A shape representing a floor is selected. (2) The parameter windowHeight is modified - all shapes on lower levels are automatically modified (3) Selecting a specific shape allows overriding the parameter on a lower level.*

Finally, all parameters of the currently selected shape or command are automatically mapped to standard GUI elements like sliders and checkboxes, seen in Figure 3.14(d). This way, all elements that are not visualized can be edited. An example workflow using this editor is shown in Section 3.4. Let us now describe those concepts in greater detail:

3D Visualization In order to edit rules and commands in a rendering, they have to be visualized. Split and repeat commands are visualized by rendering their dividing planes. We use a plane with a rounded rectangular hole in the middle (achieved with alpha tests) to reduce occlusion issues. Depth cues by reducing brightness for distant objects are employed to make distinctions of different visualized commands easier. Currently selected shapes or commands are highlighted with a surrounding transparent box. An example visualization can be seen in Figure 3.14 (b). We also experimented with visualizations of other aspects, like relationships and nesting of rules. However, we felt that a linked tree-view, as explained later, is more suitable for those aspects.

Visual 3D Editing Utilizing the visualization we allow direct editing: The user can pick a visualized element (for example a dividing plane), and drag the element around. When multiple choices are possible, a context menu allows specific selections. Internally, this works as follows: When a dividing plane is picked, we construct a perpendicular helper polygon going through the intersection location of the picking ray, as seen in Figure 3.15. We need this polygon to restrict the possible mouse positions and to fix the depth z of the intersection – without this polygon we experienced oscillations of the dividing plane during editing. The helper polygon is stored until a new plane



☑ axes[0]
☐ axes[1]
☐ axes[2]
size 0.1
☑ relative
size 0.2

Figure 3.14: *Interactive Rule Editor. (a) Tool palette allows creation of new commands and rules (button descriptions were added to the screen shot) (b) Real-time rendering of result and visualization (c) Linked tree-view. Yellow puzzle icons represent predecessor shapes, blue/green puzzle icons represent shapes occurring in a successor. (d) Parameters are automatically mapped to GUI elements.*

is picked.

When the user moves the mouse cursor, we intersect the picking ray with the helper polygon, yielding an intersection point p_i. Now we have to recalculate the parameters of the command containing the picked plane, meeting the following condition: After regeneration of the edited rule, the dividing plane has to intersect the helper polygon on the intersection point p_i.

For repeat commands, this calculation is trivial: A dot product of $(p_i - p_o)$ with the repeat axis vector v, divided by the number of the modified plane, yields the new value for the repeat with. Split commands are more difficult, because split sizes can be defined relative or absolute [MWH$^+$06]. We recalculate the parameters as follows: At first, the desired sizes d_i are calculated, this is trivially done using dot products and subtractions. Then we need to calculate split size parameters s_i that generate sizes d_i. Absolute sizes $sabs_i$ are simply set to d_i. Relative sizes $srel_i$ are set to $d_i \cdot (\sum srelold)/(scopesize - \sum sabs)$ with $srelold$ being the relative sizes before the mouse movement.

When variables are involved in the parameters, we simply add an offset to

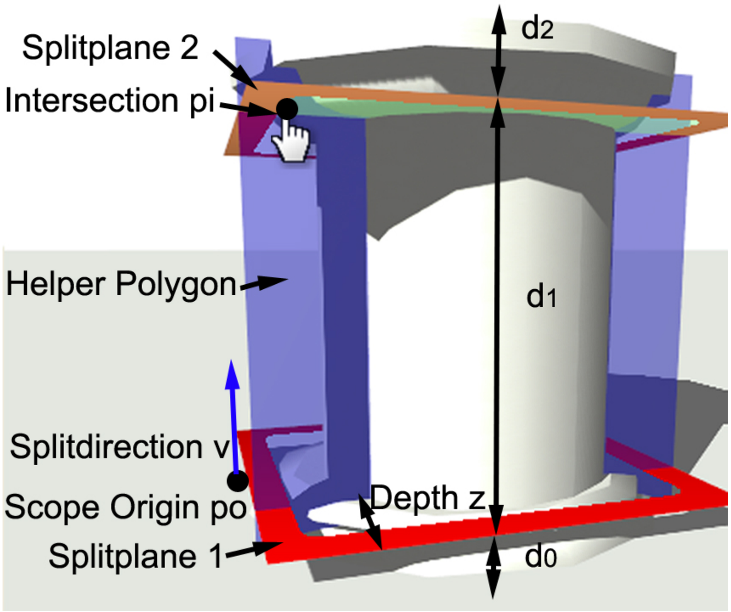


Figure 3.15: *Geometry during picking and direct dividing plane movements.*

this variable (e.g. var gets $var + 0.3$), so no variables are lost during visual editing.

Linked Views Additionally to the visualization we created a treeview displaying all rules, shapes and commands occurring in the current derivation, as seen in Figure 3.14 (c). This treeview is linked with the visual representations: Selections in the treeview automatically select an element in the derivation and the other way round. It is important to note that we have a $(1 : n)$ mapping here: One element in the treeview may correspond to n elements in the derivation, because multiple instances are possible. Therefore, when selecting an element in the treeview, just the first occurring element in the derivation is selected. Additional features of the tree-view include searching for shapes, and support for drag and drop to copy or move shapes. During evaluation, we found the treeview to be better suitable in representing relationships of rules, while the rendered result is more aimed at adjusting specific commands.

Focus and Context When editing high-level rules, for example to model building shells, visualization and rendering of lower level rules may be distracting. Therefore we implemented focusing based on the amount of levels between the edited shape u_e and other shapes u: We can set the amount of displayed levels i with a slider. Only shapes where the path between u_e and

u contains at most i shapes are displayed. For other shapes, either a proxy geometry (to represent the context) or nothing at all is rendered. An example for focusing is seen in Figure 3.1 on the right. Focusing can be separately controlled for rule rendering and visualization.

Parameter View Not all parameters of commands or shapes are mapped to visualizations. In order to enable visual editing of those parameters, they are automatically mapped to standard GUI elements, as seen in Figure 3.14(d). When variables occur in numerical parameters, slider adjustments simply add an offset to the variable.

UV Mapping Control An element missing from previous design grammars is direct control for UV texture coordinate mapping – this is very important for artists. Therefore, we introduce a new *UV mapping* command: It can be inserted into any rule, and defines a parameterized UV mapping, for example a box UV mapping with parameters tiling and offset. All shapes underlying this command in the shape hierarchy can automatically use this mapping.

Completeness of Editing Utilizing the described methods, all CGA shape concepts are visually editable: Snap lines can be inserted into a rule utilizing the tool bar buttons seen in Figure 3.14(a). They are visualized in the 3D rendering. The properties of the snap line are editable with standard GUI widgets in the parameter view. Scope modifications (translation, rotation, scale, push, pop) are also inserted using tool bar buttons, and can then either be modified using standard 3D manipulators (e.g. Arc rotate) in the 3D view or by modifying sliders. When a rule is selected, a text field in the parameter view allows entering arbitrary conditions using variables and Boolean operators. As occlusions are also defined as conditions (using automatically initialized variables like `isShapeOccluded`), they can be defined analogously.

3.4 Implementation and Results

Modeling Workflow As our main contributions enable complete visual editing of building grammars, we will now show a case study creating a building grammar from scratch. In Figure 3.16 the necessary steps to create a simple building in 3.5 minutes are laid out (please refer to the video for a real-time capture of the entire session). (1) A few empty rules are created and named. (2) Selecting a rule and clicking on the split icons allows easy splitting of shapes. (3) Drag and drop allows easy definitions of shapes to be used as in the split command. (4) Adding terminal shapes allows geometry addition. Textures are dragged from a texture library. Cut and paste of shapes allows fast setting of the wall tiles. (5) Direct dragging of split planes is possible in the 3D view. When `CTRL` is pressed during dragging, the planes are distributed symmetrically. Extrusion of the window can be done with one mouse click. (6) Selecting the high-level rule `house`, adding a component split with a simple click and adding a repeat command creates the building shell. Please note that both top-down and bottom-up modeling approaches can be easily combined with our method. (7) Repeat commands are added to distribute windows. (8) The `winwall` rule is dragged into the repeat command. (9) Simple (but automatically adapting) roofs can be added as a terminal shape.

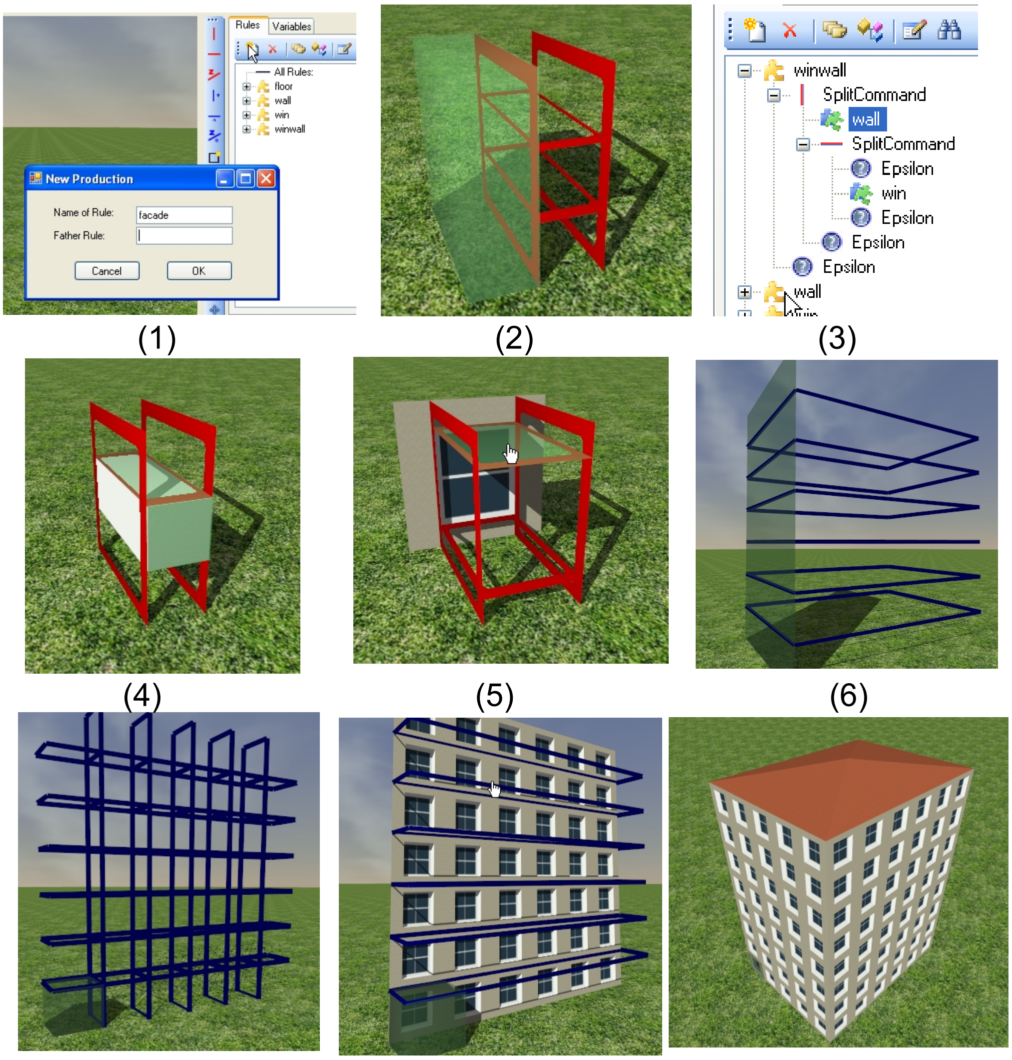
Starting from this simple building, we were able to create the more complex building seen in Figure 3.12, which has many ornamentation details, in an additional 10 minutes.

Usability Our method allows visual editing without resorting to text files. The only necessary textual entries are naming of new rules and parameters. Drag and drop is used in many places, simplifying rule creation. For example the simple building in Figure 3.16 corresponds to around 55 lines of CGA shape code. We therefore feel that this method substantially simplifies modeling of grammar-based architecture.

In order to actually evaluate the usability, we introduced artists from our industry partner to our tool. After a short introduction we asked them to create a building, and document the difficulties arising during modeling. As an advantage of our method, they found visual editing of split rules to be much more intuitive than textual editing. Especially the feature to mirror split sizes during editing by pressing a specific key was considered to be very useful.

Two identified shortcomings stem directly from the rule-based nature of CGA shape: First, the dependency between rules is not clearly visible. This can make it hard to understand the rules created by a different artist. The artists proposed to use a node-based display of rules in order to alleviate this. We will incorporate this in future work. Second, artists commented that they would be very exited about alternatives to rule editing by editing buildings using a philosophy of modeling by example.

A number of suggestions was related to software engineering issues and the best way on how to incorporate the software in the existing modeling pipeline. Concerning the expressivity of CGA shape, the artists suggested to add more control to the high-level building shape, mainly by allowing



Name of Rule: facade
Father Rule:

(1) (2) (3)

(4) (5) (6)

(7) (8) (9)

Figure 3.16: *Example workflow using our rule editor. Creating this simple building required 3.5 minutes. These are screenshots from an recorded user interaction. The screenshots were cropped to magnify important details. A description of the individual screenshots is provided in Section 3.4. In approximately additional 10 minutes we were able to create the building seen in Figure 3.12 which has many ornamentation details.*

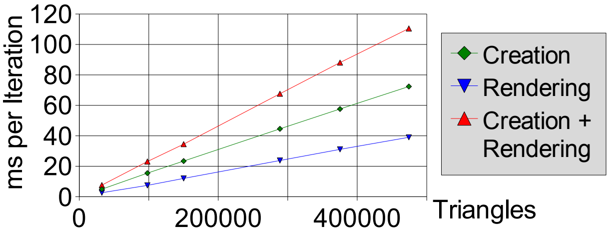


Figure 3.17: *Milliseconds per iteration versus triangles for various modes.*

arbitrary ground-plans unique to every floor.

Performance One important aspect of a visual editing system that was not mentioned so far is response time. Our visual editor frequently needs to regenerate a whole instance from the rulebase, for example whenever a variable is changed or a constraint like the floorplan is modified through continuous dragging. Thus the performance of the grammar system is as important as the rendering performance itself. Previous work has reported building generations times in the order of a few seconds, which is far too low for interactive (i.e., ¿ 20 updates per second) manipulation. In our implementation, we used several optimizations to ensure interactive generation and rendering times, including custom memory manager, lists, sorting algorithm and random number generator. For the occlusion test required in [MWH$^+$06] to prevent intersections of shapes, we employed the hardware-accelerated algorithm proposed in [Kno03]. This is a combination of stencil buffering and hardware occlusion queries, where a building is interpreted as a shadow volume, and the z-Pass algorithm is used to search for shapes intersecting this volume. This is significantly faster than the octree-based method originally proposed.

In Figure 3.17 we can see that interactive performance for rendering is achieved for all tested buildings even on a relatively slow PC with Athlon XP 2600 CPU, 1024 MB RAM, and Geforce 6600 graphics card. In combination, creation and rendering is still interactive for a building with about 200,000 triangles. Figure 3.17 also shows that the performance scales linearly with the building complexity. This is beneficial, as a linear scaling provides a cushion for more complex buildings. For this test, the building in Figure 3.1 on the middle right was varied using a global height parameter.

Limitations We tried to experiment with tree modeling with reasonable success. We believe that our current implementation is mainly suitable for buildings, but we intend to experiment with a larger class of objects in future work. We expect that we would have to extend the rule set, but we believe that many fundamental principles of visual editing of grammars could be reused. Some restrictions in our current implementation stem from choosing CGA shape as underlying production system: CGA shape has no direct support for curved surfaces, making bridges or complex mechanical shapes hard to generate. However, when CGA shape is extended to support these concepts, our approach should be able to handle them seamlessly.

3.5 Comparison to Related Work

In the context of L-systems, positional information was used by querying functions depending on the current turtle position [PMKL01]. This allows determining high-level shapes of plants. Currently such functionality is not present in CGA shape, although it would be interesting to explore how this could be applicable to control high-level façade structures in future work. Also, methods to graphically model plants were introduced: In [LD99] components specifically targeted at plant modeling are connected in a graph. As those components are very specific to plants, this system can not be directly used for architecture. In [BPF$^+$03] a multiscale representation of plants is used in order to minimize the total number of parameters needed in order to specify a plant. Their system does *not* always maintain persistence of plant modifications, for example when the branch density is increased, modifications to specific branches can get lost [BPF$^+$03]. The persistence problem for parametric modeling was described in [Sha02, HJA02, Hav05], and it was pointed out that there is no general solution known at this time [Sha02]. Our solution to the persistence problem is specifically targeted at production systems, not parametric modeling in general. We achieve this by exploiting the production hierarchy and semantic informations.

Two methods were proposed that allow creation of procedural architecture without text editing: First, an image-based approach to create CGA shape grammars [MZWG07], which allows creating rules from building images. Our approach could be well employed to visually enhance or correct the deduced rulebase. Further, a framework for procedural modeling using a visual language was introduced [GK07]. Essentially, a mapping of programming constructs to visual symbols is performed. Those symbols can be combined visually. In contrast to our approach, *no* grammar or rulebase is used. Therefore we think this approach is orthogonal to ours and could be combined: The visual language could create some building parts, while our method could visually handle everything rule and grammar related.

A similar argument holds for other techniques *not* based on grammars: [Hav05] introduces modeling using a stack-based programming language.

[BBJ$^+$01] was shown to be applicable to building parts. Persistent building interior generation was discussed in [HBW06].

3.6 Conclusion and Future Work

We present the first real-time visual editing system that allows an artist to visually create a rulebase for shape grammars from scratch. Furthermore, we extend previous shape grammar approaches by providing direct local artist control over the generated instances, avoiding combinatorial explosion of grammar rules for modifications that should not affect all instances. This effectively combines the power of procedural modeling techniques and standard 3D modeling tools. We have described the selection and persistence problem, and provided a solution using so-called instance locators.

While our framework enables real-time editing of individual buildings, there are still some open problems if those buildings should be used in a real-time game. These include: (1) Automatic LOD generation: In order to render large scale scenes, low-detail versions of the building should be created automatically, assisted by the grammar structure. (2) Mesh cleanup: At the moment, the mesh may contain T-Vertices and coplanar polygons, which needs to be resolved for real-time rendering applications. For our visual editor, the next logical step is to extend the concept to whole cities, not just individual building instances, as we will show in the next chapter.

4

Direct Artist Control for Procedural City Layouts

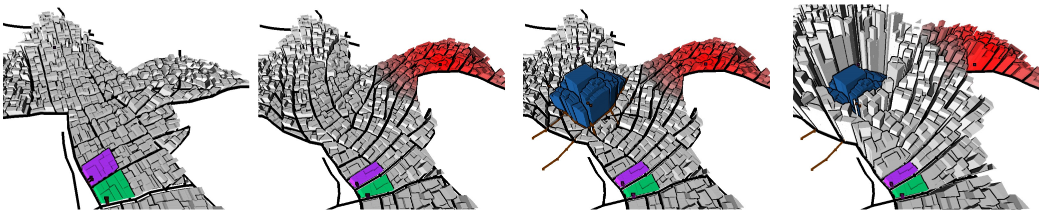


Figure 4.1: *This figure illustrates possible urban transformations using our method. From left to right: (1) The initial urban layout. (2) The layout is transformed using a rotation with a soft influence radius, and a red color assignment with a soft border region is added. (3) A different city center is merged into the layout. (4) A height assignment is modified. Note that during all those steps, the color and height of the two distinct buildings on the bottom stayed persistent, because they had individual assignments.*

In the previous chapter, we provided solutions for persistent direct artist control and visual editing for building generation. However, those solutions are not directly applicable to urban layouts, mainly because they do not have a strict hierarchy like buildings. This makes the usage of the introduced locators difficult.

Therefore we will now introduce novel solutions for this problem targeted at urban layouts. A few examples of this solution are show in Figure 4.1. The results of this chapter are currently under review for a conference publication.

Let us now explain the problems of visual and direct artist control in the context of city modeling: Procedural city modeling is a rapidly evolving field in computer graphics, with applications to urban planning, design reviews, game design and others. Procedural techniques are often based on grammars and parameters, which makes it difficult to achieve exactly the desired outputs. City layouts are responsible for the overall aspect of an urban model and for controlling all the other parts of a procedural city generation system, and therefore it is paramount that the user has a powerful control mechanism for this step.

Current urban layouting systems [WMWG09, Pro10] offer only limited editing possibilities once an urban layout has been produced procedurally. While it is possible to drag individual nodes in the street graph, the resulting urban layout has intersections and is therefore not *valid* anymore (see Figure 4.2). To make the layout valid requires expensive manual operations like filling the gaps, reconnecting the streetgraph when an element is deleted, making space for new elements, etc. Even worse, if the underlying procedural description changes, the whole layout has to be regenerated from scratch, losing all manual customizations since they are not persistent. More complex operations like consistently merging different layouts, possibly from different sources (procedural or manual), are practically impossible.

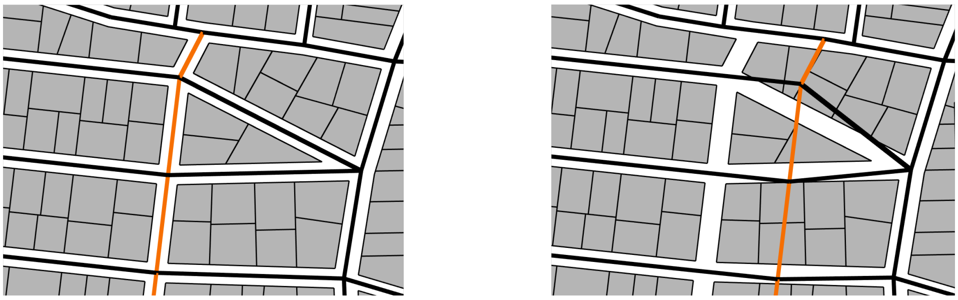


Figure 4.2: *This figure illustrates how, using traditional urban layouting tools, a simple translation transforms a valid layout (left) into an invalid one with intersections (right).*

In this chapter we present an interactive city modeling system that is built on persistent editing operations that remain in the space of valid urban layouts. The system combines procedural edits, local manual edits, higher level manual edits, as shown in Figure 4.1, It is designed to meet the following research challenges:

Direct control and editing of procedural layouts. Ideally, to modify an urban layout, designers would like to use simple and intuitive editing operations, like semantic selection of elements, drag and drop, insertion and deletion of elements etc. Most importantly, these operations should again produce a valid urban layout. They need to handle *changes in topology*, and should be designed to have *local influence* only. In particular, they should be *reversible*, so that the original appearance of a city part is restored when an inserted element gets dragged to another location (this is also called *circular editing*). An example of such an operation supported by our method is shown in Figure

4.3.

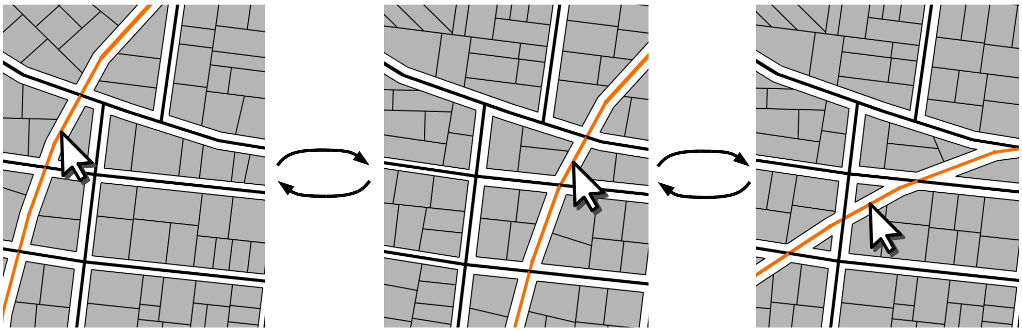


Figure 4.3: *The orange street is moved and rotated. The underlying parcels update accordingly. When the street is moved or rotated back, the original layout is regained again, providing circular editing capabilities.*

Combining urban layouts. One of the most frequent problems that occurs in urban modeling is to consistently merge urban layouts at different levels. Examples include inserting a (manually modeled) parking lot, a park, a whole street, a whole block (like a shopping mall) or even a whole quarter into an existing city layout. Consistent merging capabilities allow designers to reuse components or to model components separately. There is currently no solution to merge urban layouts automatically. One such merge process possible with our system is shown in Figure 4.4.

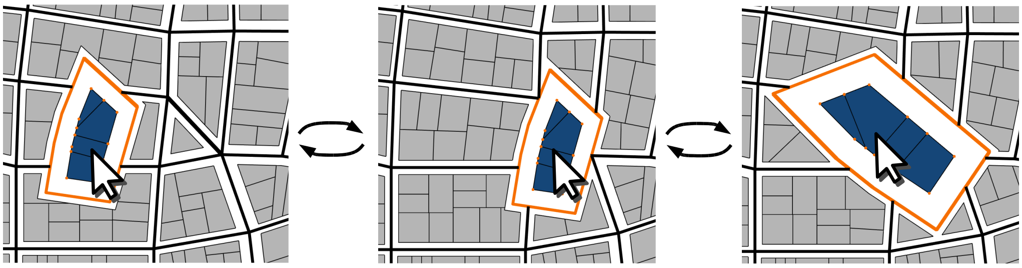


Figure 4.4: *Content from a different source, highlighted in orange, is inserted into the layout and moved, scaled and rotated. Full circular editing is supported.*

Persistence. Changes applied by the user should survive local and global editing operations. Urban layouting drives the whole city generation, like parameter assignments, distribution of landmarks etc., so these assignments need to survive modifications of the urban layout. Modifications to an already customized urban layout belong to the most expensive design operations for example in level design.

Main contributions In order to meet the research challenges previously listed, this chapter introduces a new set of editing operations that transform one valid urban layout into another valid one. Full *circular editing* capabilities like drag and drop, insertion, deletion etc., with arbitrary topological changes are provided. The operations are based on the combination of a *layering system* in the spirit of image manipulation programs, and a novel *merging algorithm* that consistently merges urban layouts based on graph cuts. We also extend the locator concept introduced in previous work to achieve *persistent anchored assignments*, linked to elements in an urban layout, allowing modifications to survive global procedural modifications. These methods are implemented in a city modeling system that combines the power and convenience of procedural street generation with the flexibility and direct artistic control of a traditional content creation system.

4.1 Transformations of Urban Layouts

The main problem to solve in a city modeling system is how to transform a *valid* urban layout into another one, as illustrated in Figure 4.2. For this we introduce three basic transformation operators and show how they can express most direct urban layout editing operations.

4.1.1 Definition of Urban Layouts

An *urban layout* U consists of a *street network* and *parcels*, as shown in Figure 4.5. The street network is given as an undirected planar graph $G = (V, E)$ with nodes V and edges E. We also refer to the nodes as crossings and the edges as streets (note that for simplicity we do not distinguish between street segments and streets). A parcel $p \in P$ is a possibly concave, simple polygon with at least 3 vertices. When referring to either a street, a crossing or a parcel we use the term *element*.

A face of an embedded planar graph is a cycle that surrounds a region that contains no edge. We call the faces of the planar graph G the *blocks* B of an urban layout. The faces obtained when ignoring all minor streets are called *quarters*.

Street network and parcels are connected through a binary ownership relation $O \subseteq P \times B$ between parcels and blocks: Every parcel p that is completely inside a block b is *owned* by this block, and $(p, b) \in O$. A parcel is owned by at most one block. Parcels without an owner are called *free* parcels, while the others are called *enclosed* parcels. Given an urban layout, O can be

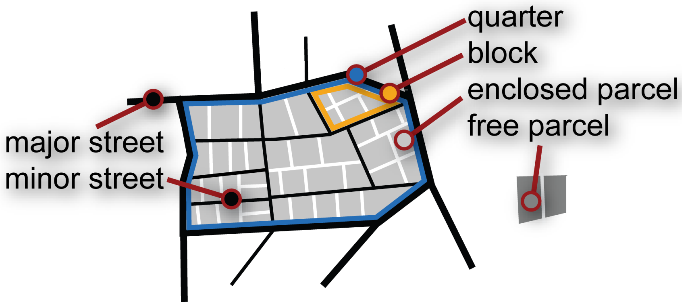


Figure 4.5: *The basic building blocks for urban layouts.*

calculated by first finding all blocks and then performing containment tests of all parcels with those blocks.

Crossings, streets and parcels can have an arbitrary amount of key-value pairs attached, we refer to them as *tags*. The tags of every street must at least contain the key `streetType` with the value of either `minor` or `major` to discern between minor and major streets (other streettypes like highway would also be possible).

The definition of urban layouts so far does not ensure that such layouts make sense. It allows streets intersecting without crossings, parcels intersecting streets etc. In order to restrict editing operations to "useful" urban layouts, we define an urban layout U as being *valid* if (1) there are no intersections between streets and (2) parcels do not intersect streets or other parcels. We denote such a layout with $\overline{U}$.

In the following, we define three operations on valid urban layouts $\overline{U}$, i.e., the result will again be a valid urban layout. The first operation is the non-topological transform T. The second operation is a flexible binary merge operation $M_f(\overline{U}_a, \overline{U}_b)$ based on graph cut, which produces a new valid urban layout from two input layouts. The third operation $M_h(\overline{U}_a, \overline{U}_b)$ is a special case of the flexible merge. In Section 4.2 we will then describe how most direct editing operations can be expressed using T, M_f, M_h and a layering system.

4.1.2 Non-Topological Transform

Many editing operations are small changes to an existing layout. For example, the user drags one node, denoted v, in the streetgraph by a small amount, represented by the affine transformation A. As a *small* transformation we define one that does not change the topology of the streetgraph (i.e., the dragged street does not intersect a new street). For such small transformations, the user expects the layout to remain valid (i.e., the parcels in adjacent blocks move with the changed street), and that these changes have as few side effects as possible.

Therefore we introduce the non-topological transform $T(\overline{U}, A, W)$, where A is an affine transformation and W contains a weight $w_v \in [0, 1]$ for every node in $v \in \overline{U}$. In the previously mentioned example of a dragged node v, A would be a translation and the weights would be 1 for v and 0 for the other nodes. $T(\overline{U}, A, W)$ ensures that after applying A the layout is still valid, and works as follows:

(1) Every node $v \in V$ is transformed using $v_1 = (vA)w_v + v(1.0 - w_v)$. This creates a potentially invalid layout U_1. (2) We test U_1 for street to street intersections, to ensure that there are no topological changes. If there are intersections, $T(\overline{U}, A, W)$ simply returns the original layout $\overline{U}$, essentially ignoring the transformation A. (3) If there are no street to street intersections, we update the parcels in U_1 to ensure there are no street to parcel intersections. This creates a valid layout $\overline{U}_2$, which is then returned. Note that concurrently to our work, the newest release of the commercial Cityengine [Pro10] implemented some kind of automatic parcel updating, however they did not publish any details on how they achieve this, so we can not provide a comparison to their method. Our method of updating the parcels has several steps:

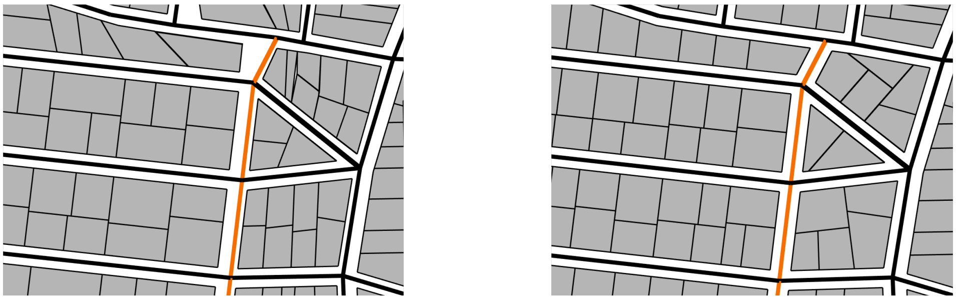


Figure 4.6: *The orange street was moved. Left: Smooth transformation of parcels, exhibiting some distortions. Right: Local parcel regeneration.*

Finding affected parcels All parcels contained in blocks adjacent to transformed streets need to be updated. The algorithm goes through all blocks and checks whether any of the vertices v contained in the cycle defining the block has a non-zero weight w_v. In turn, all affected parcels contained in those blocks are found using the parcel ownership relation O. We denote blocks before applying the transformation A as b and after applying A as b_1.

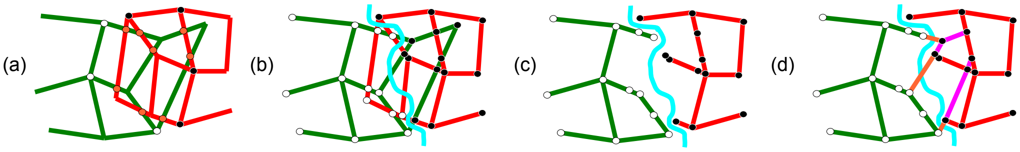


Figure 4.7: *Application of graph cut to city layouts. (a) Creation of shared graph. Green:* $\overline{U}_a$*, red:* $\overline{U}_b$*, orange dots: intersections, white/black dots: constraint arcs. (b) The blue line represents a possible cut. White and black dots now represent the graph coloring. (c) Deletion of streets with nodes in wrong partition. (d) Mending of holes by including certain streets of* $\overline{U}_a$*.*

Parcel update It is desirable to update parcels in a "smooth" way. However, larger deformations require adding or removing parcels. We therefore provide two mechanisms for parcel updates: a *smooth transform* that geometrically distorts the parcels to fit the new block, and *local regeneration*, which procedurally recreates the parcels in the block. The differences are shown in Figure 4.6. We decide on a per-block basis which mechanism to employ: When either the area of the corresponding block or the angles enclosed between connected block edges change more than user-defined thresholds, the parcels of this block are regenerated, otherwise they are transformed smoothly.

Smooth transform We first calculate the mean value coordinates [HF06] of every vertex of every affected parcel with respect to its untransformed owner block b. Using those coordinates, we recalculate all vertex positions with respect to the transformed block b_1. This method retains the original parcel layout, but may lead to distortions.

Local regeneration If the distortion through a smooth transform becomes too large, it is better to regenerate affected parcels: First, all parcels originally belonging to b are deleted. The new parcel boundary is obtained by *shrinking* b_1 in order to accommodate for the distance of the parcels to the road. Shrinking arbitrary polygons is a non-trivial problem, and we employ the skeleton-based algorithm found in CGAL [Cac09]. Finally, parcels are generated procedurally in the new parcel boundary using the method introduced by Weber et al. [WMWG09].

Regeneration has the advantage that even for larger transformations, correctly sized parcels are generated. As a disadvantage, the original parcel layout of the block is destroyed, as the newly generated parcels may not match the deleted ones. However, customizations to parcels can still be preserved using anchored assignments as described in Section 4.3.

4.1.3 Flexible Merging using Graphcut

Editing operations that change the topology of the street network are much harder to realize than topology-preserving ones. An important contribution of this chapter is that we express these edits using sequences of operations that involve *merging* two different urban layouts, as will be discussed in Section 4.2. As a simple example, an arbitrary translation of a street can be done by moving the street to a separate layout, translate the street there, and merging the temporary layout back. More involved operations require the merging of whole city parts with user-defined priority maps.

As the heart of these operations, we introduce a flexible binary merge operator $M_f(\overline{U}_a, \overline{U}_b)$ that is designed to merge two urban layouts $\overline{U}_a$ and $\overline{U}_b$, producing a new valid urban layout. Similar to alpha mattes in image processing, we allow the artist to flexibly assign priorities to elements in the layouts.

Unlike image mattes, the priority cannot be incorporated through a simple compositing operation of regular images. Instead, in this section, we show a compositing algorithm on urban layouts that is computed using graph cuts.

Let us first review graph cuts [FF62]: Consider a graph $G = (V, E)$, a source $s \in V$, a sink $t \in V$ and a *capacity* c_e associated with every edge $e \in E$. Then an s-t graph cut partitions the vertices into two subsets S and T with $s \in S$ and $t \in T$. The cut-set includes all edges whose vertices are in different partitions. The cut is minimal if the sum of all edge capacities in the cut set is minimal.

In image processing, a graph cut is often employed to merge different images [KSE$^+$03] when blending is not desirable. The question now is how to cast the merging of two urban layouts $\overline{U}_a$ and $\overline{U}_b$ into a graph cut problem. The general idea is to work on the street networks of the layouts, and interpret the user priorities as capacities for the graph cut.

However, there is an additional challenge: In order to calculate a graph cut, a single shared graph needs to be constructed from the two source layouts. In image processing this is straight forward, as the different images share a common pixel grid which defines a shared graph. There is no obvious shared graph for the city layouts $\overline{U}_a$ and $\overline{U}_b$. Further challenges are the creation of source and sink nodes, and the reconnection of the two partitions. The whole procedure works as follows, and is illustrated in Figure 4.7:

(1) Assign priorities (2) Create a shared graph. (3) Automatically create a source and a sink, and create constraint arcs to them. (4) Search for a minimal s-t cut. (5) Delete streets that are in the wrong partition, and reconnect partitions. (6) Update the corresponding parcels.

Assign priorities Numerical priority assignments to nodes are done using anchored assignments as shown in Section 4.3, giving two separate priority distributions for $\overline{U}_a$ and $\overline{U}_b$. Each street samples the corresponding priority distribution at its midpoint to obtain its capacity c_e to be used for the graph cut.

Creation of a shared graph We need to bring $\overline{U}_a$ and $\overline{U}_b$ into a common shared graph G in order to apply the graph cut algorithm: First, every street from $\overline{U}_a$ is copied to the shared graph G. Then, an intersection test of every street in $\overline{U}_b$ with the streets in G is performed (coincident streets are deleted). A new crossing is created at every intersection, and the streets involved in the intersection are rerouted along those new crossings. We call those new crossings *intersection nodes* N_i. During insertions we add a tag to every street indicating its original layer. An example shared graph is shown in Figure 4.7(a). To provide better numerical stability we remove all dead end streets.

Source and sink connections We create a source node corresponding to layer $\overline{U}_a$, and a sink node for layer $\overline{U}_b$. While source and sink do not have a position in space, it is crucial which nodes are connected to source and sink: The minimal cut should be located in the region where $\overline{U}_a$ and $\overline{U}_b$ overlap. If it was outside, large holes would occur after step 5 in the algorithm (see grey lines in Figure 4.8 for such cuts). Therefore, analogously to image merging [KSE+03], the borders of the overlapping region need to be constrained to belong to one of the partitions, preventing the cut to go through the border. This is done by connecting each border node to source or sink using a constraint arc, e.g., an edge with a high capacity, which essentially forces the node to be in the source or sink partition after the cut.

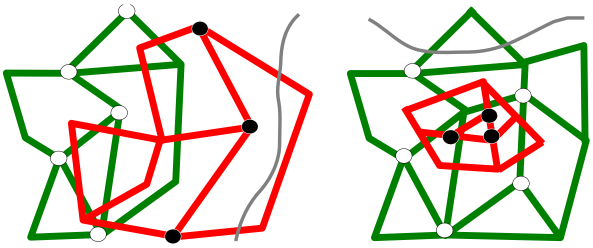


Figure 4.8: *Visualization of constraint arcs: White and black dots represent necessary constraint arcs to the source and sink respectively. Left: Overlapping region. Right: Red graph is completely contained in the green graph. Grey lines are examples of cuts that we want to prevent.*

This is simple for regular images, but for irregular street networks we need to employ a more involved algorithm to find border nodes: (1) Find potential border edges, which connect one node a from $\overline{U}_a$ or b from $\overline{U}_b$ with an intersection node from N_i. All nodes a and b are potential border nodes. (2) Remove all potential border nodes a that are inside a block of $\overline{U}_b$, and all nodes b inside a block of $\overline{U}_a$. This removes all nodes that can not be a border because they are located inside a block of the other graph. (3) All remaining nodes are border nodes. An example set of border nodes is shown in Figure 4.8 (left).

There is one special case, shown in Figure 4.8 (right), that this algorithm does not handle: When the graph $\overline{U}_a$ is completely inside the graph $\overline{U}_b$, no border nodes will be found for $\overline{U}_a$. To still constrain some nodes of $\overline{U}_a$, we use the following heuristic: The nodes of the edges with the n highest priorities are constrained. We found $n = 5$ to be sufficient in our test cases.

Computing minimal cut The algorithm of Edmonds and Karp [EK72] is used to solve for the maximum flow and respectively the minimum cut. The result is a coloring of the graph where white nodes correspond to the source partition and black nodes to the sink partition, as shown in Figure 4.7(b).

Deletion of streets and reconnection Now we delete all streets whose nodes are not in the correct partitions. $\overline{U}_a$ nodes should be in the white partition, $\overline{U}_b$ nodes in the black one. This separates the graph as seen in Figure 4.7(c). In order to mend the created holes, we first add back all streets of $\overline{U}_a$ that have at least one correct node, shown as orange lines in Figure 4.7(d). This may still leave some holes. Therefore we start a depth first search at each border street of $\overline{U}_a$ and add back all encountered $\overline{U}_a$ streets until a street would enter a block of $\overline{U}_b$. All streets added back this way are shown in magenta in Figure 4.7(d).

Incorporation of parcels To correctly handle parcels, the following preparation steps are performed before the shared graph is created: First, all free parcels are enclosed with streets. This is necessary because the merging only considers the streets. Then, all blocks B_a, B_b and ownership relationships O_a, O_b of the layouts $\overline{U}_a$ and $\overline{U}_b$ are calculated.

Using those relations, adding the parcels back after the graph cut is done as follows: First we find all blocks of the graph cut result. Then, for every block that contains only streets of one layer, the parcels that were previously defined for this block are found using the ownership relation and are added. The parcels for all the other blocks are procedurally regenerated.

4.1.4 Hard Topological Merge

One important special case of the flexible merge $M_f(\overline{U}_a, \overline{U}_b)$ occurs when the priorities of $\overline{U}_b$ are much higher than the priorities of $\overline{U}_a$: All the elements of $\overline{U}_b$ will be present in the result. This has two advantages: First, having all the elements of $\overline{U}_b$ retained may be the desired result when an artist merges a small but important element into a city. Second, we can significantly speed up this special case, as we will show in this section.

Let us therefore introduce the *hard* topological merge operator $M_h(\overline{U}_a, \overline{U}_b)$, with the property that all elements of $\overline{U}_b$ are present in the result. The main idea to speed this up is that we already know where the minimal graph cut should be: All streets of $\overline{U}_a$ that intersect the concave hull of any connected component of $\overline{U}_b$ must be in the cut set. This ensures that the cut is just outside of $\overline{U}_b$.

As we know where the cut should be, we do not need to perform a graph cut. But there is another chance for improving performance: In the first step of the flexible merging, a shared graph is created by intersecting every street of $\overline{U}_b$ with the streets of $\overline{U}_a$. However, as we know that no street of $\overline{U}_a$ should protrude into a concave hull of $\overline{U}_b$, we can simply intersect those concave hulls with $\overline{U}_a$, reducing the amount of intersection tests. This simplifies the algorithm to the following:

(1) Insert all elements of $\overline{U}_b$ into the result. (2) Find the concave hulls of connected components. (3) Clip all elements of $\overline{U}_a$ against the concave hulls, and insert the clipped result. (4) Incorporate the parcels.

Step 1 is trivial, and step 4 is the same as in the flexible merge, therefore only steps 2 and 3 will be explained now.

Finding and clipping against the concave hulls When clipping against a concave hull, numerical problems can occur when the hull contains streets that do not enclose any block. Therefore we find those non-enclosing streets and handle them separately. An example classification is shown in Figure 4.9.

Finding the hulls and the non-enclosing streets can be done using the block list B_b and the ownership relation O_b. All streets that are not a border of any block are non-enclosing streets. All streets that are a border of exactly one non-empty block are part of a concave hull without dead-ends. All concave hulls can now be found by using the block finding algorithm while ignoring all streets that are not part of the concave hull.

The actual clipping is trivial: For the non-enclosing streets, line intersections are performed and new crossings are inserted at intersections. The concave hulls, on the other hand, are interpreted as polygons and a line-polygon clipping algorithm is employed, inserting new crossings at intersections on

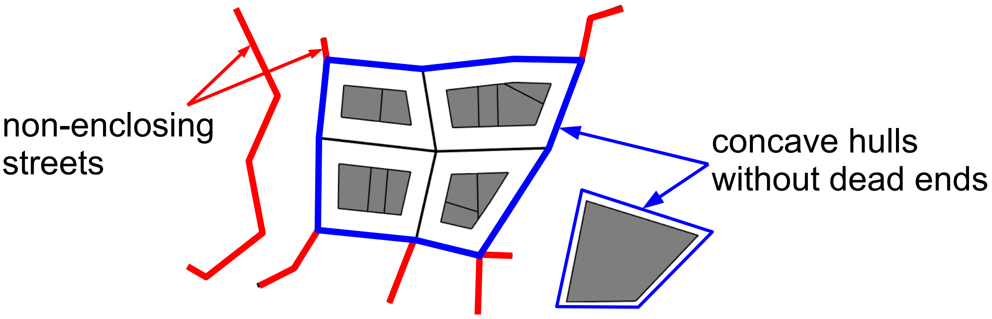


Figure 4.9: *Streets are classified into hull and non-enclosing streets for numerically stable clipping.*

the border.

4.2 Editing Operations Using Layers and Layout Transformations

In this section we introduce our layering concept and explain how most editing operations can be mapped to a combination of layers and the operators introduced above. All operations result in valid urban layouts.

4.2.1 Layers

Layering is well known in commercial image processing tools like Adobe® Photoshop™, where it is used to arrange, merge or protect content. For urban layouts the merging process to get one final result from multiple layers is much more involved. While in image processing merging can be trivially done using blending with transparency values, in urban layouts streets or parcels may intersect elements in a different layer, making a rerouting or deletion of streets necessary. For this complex operation we use the operator M_f or M_h described above, allowing the artist to specify priority distributions on layers similar to alpha mattes in image processing tools. Note that while the commercially available Cityengine [Pro10] supports defining content on multiple layers, they do not provide any means of merging layers or guaranteeing valid layouts.

Layer definition A layer L consists of one urban layout, i.e., a street network and parcels. A scene can have a finite number of layers $(L_1, L_2, \ldots, L_n)$. What is finally displayed, exported, etc., in an interactive editing system is a merged layer L_m which is iteratively defined using $L_m = L_n$ and continues with $L_m = M_f(L_{n-i}, L_m), i = 1 \dots n-1$, until all layers are included. This definition ensures that elements in layers with a higher number precede elements on lower layers. If an artist wants to reduce the amount of layers, two layers can be collapsed to one combined layer using M_f or M_h.

4.2.2 Basic Editing Operations

The basic editing workflow involves *selecting* elements in an urban layout and *transforming* them.

Simple, soft and semantic selections Selections of elements are represented as a weight $\in [0, 1]$ for every element. A user can select *single* elements or a region using mouse clicks. This sets the weights for all selected elements to 1 and the other ones to 0. We also provide a soft-selection tool: When the user clicks on a city part, the weights of every city element are set according to the distance to the mouse position. Selections can also be *semantic*: when a single street is selected, the selection can automatically be extended to the continuation of this street. At both end nodes of the street s we look at all connected streets s_c that share the same street type. Then we calculate the angles α_c between the incoming street s and the outgoing streets s_c. When $|180° - \alpha_c|$ is higher than a user defined threshold, the street s_c is not considered further. We set the weight of street s_c with the smallest value of $|180° - \alpha_c|$ to 1, and repeat the whole process recursively until no further streets can be added. The result is visualized in Figure 4.10.

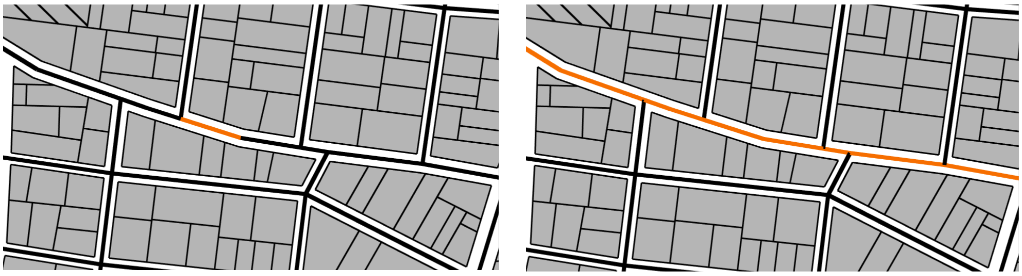


Figure 4.10: *Semantic street selection: First, one street is selected, as shown in orange on the left. The continuation, shown on the right, is then automatically found by walking along the streets and comparing angles.*

Geometric transformations Affine transformations, denoted as A, like translation, rotation and scaling can be applied to a selection. When the artist does not want topological changes to occur, this can be represented as a non-topological transformation $T(\overline{U}, A, W)$ of an urban layout as discussed in Section 4.1.2, where W represents the selection weights.

Topological changes If the artist wants to have topological changes while still retaining cyclic editing capabilities, we allow this using the following flexible and generic way: The artist can delete the selected elements in L_a and insert them into a new layer L_b, and apply the transformations to L_b. Now an arbitrary amount of cyclic editing operations can be performed in L_b, as shown in Figure 4.3 and 4.4, without modifying L_a (except for the initial deletion of the selected elements). Through the layering system, they will be merged at the new position with the current layout, always giving a valid urban layout as result.

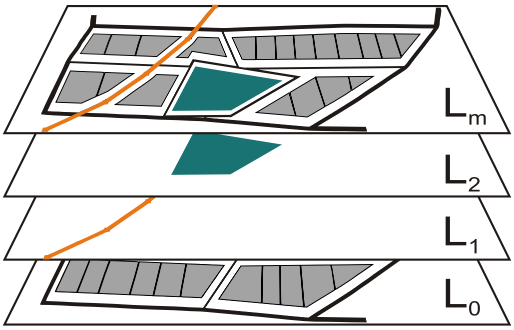


Figure 4.11: *Layering: An existing urban layout in layer L_0 is merged with a street on layer L_1 and with a park on layer L_2 resulting in the merged layer L_m.*

Insertion Inserting a new element (street, parcel, ...) works by placing it on a new layer L_o. Through the definition of the layering system, L_o will be automatically merged with the existing layers to give a merged result L_m. The element can also be transformed geometrically (dragged, scaled, rotated) to find a good place, while the merged layer L_m gives interactive feedback on the result (see Figure 4.11).

Deletion Upon deletion, the selected elements are simply removed from the urban layout. Since no new intersections are created, the resulting layout is valid.

4.2.3 Further Examples of Direct Artistic Control Using Layers and Merging

Persistent local changes In procedural editing systems, changing the input parameters to the procedural algorithm usually causes a regeneration of parts of or the whole urban layout. In order to protect local modifications to the urban layout, the user can mark important elements with the key-value pair $protected = true$. Now, before the regeneration occurs, the system automatically copies all marked elements into a new higher level layer. Then the urban layout in the original layer is replaced with the newly generated one, and the merging process ensures that the protected elements are retained. Figure 4.12 shows an example of this process.

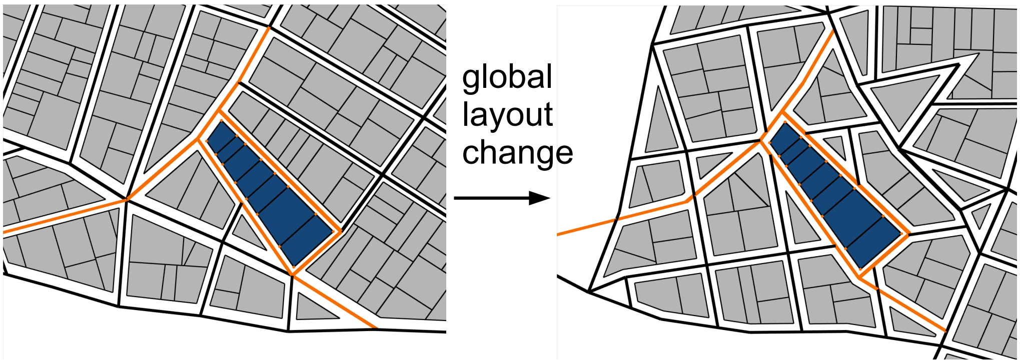


Figure 4.12: *Left: The user marked the orange elements as protected. Right: After a global layout change, the elements are still preserved.*

Merging assets It does not matter if the urban layout in a layer originated from a procedural creation algorithm or was hand crafted by a designer. The layering system allows merging content from different sources in a unified way. As an additional benefit, a procedural algorithm does not need to know anything about the layering system, and thus every street generating algorithm can be employed.

Combining styles using flexible merging When street networks with different styles are defined in different layers, an assignment can be used to specify where a specific style should be used. An example of this is shown in the accompanying video.

Tweaking in an advanced development stage In the context of computer games, moving gameplay relevant urban layout parts into a distinct layer allows moving them freely around the city. This enables fine tuning during the whole production process.

4.3 Persistent Anchored Assignments

In a city modeling system, urban layouts are refined by assigning tags (key/-value pairs) like building height to elements in the layout. In previous work, this is either done using global image maps [PM01] or by directly modifying tags of individual elements. However, when using our flexible editing operators, those methods would not be appropriate: A global image map would not follow geometric transformation, while direct assignments to elements would be lost when the elements are deleted (for example caused by a global regeneration), causing a persistence issue.

Therefore we introduce *anchored assignments*, which follow geometric transformations and are persistent after element deletions. An example is shown in Figure 4.13. In the following sections we will first define them and then show how they are used.

4.3.1 Definition of Anchored Assignments

One anchored assignment consists of *tags*, a *target*, a *world space position* denoted as Pos, and an *anchor*.

Tags describe *what* key/value pairs to assign, and can represent properties like building height, type or style. They can also point to specific 3d assets to be used, enabling the placement of landmark buildings. Also priority assignments for our flexible merge operator can be represented in tags.

The world space position is a simple vector representing the global placement of the assignment in the city.

A target specifies to *which elements* the tags should be assigned. It contains a list of element types that should be affected, including minor/major streets, nodes or parcels. Further, it specifies if only the nearest element of given type to Pos or a complete region centered at Pos should be affected. For regions, it is also possible to specify a numerical distribution, for example a radial falloff with respect to the distance to Pos. This is especially useful to define priority distributions for flexible merging.

An anchor describes how Pos should react to transformations of the layout. It contains a pointer to one specific element of the layout. When this element is transformed, the same transformation is applied to Pos, essentially moving the assignment relative to the element.

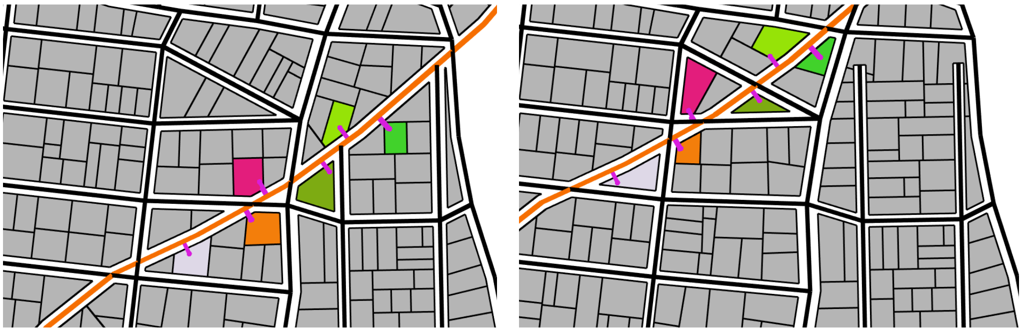


Figure 4.13: *Purple lines represent a color assignment anchored to the adjacent street, with a parcel as target. When the orange street is moved, the anchored assignments stay relative to the street, and apply to the nearest parcel.*

4.3.2 Usage of Anchored Assignments

We will now show how to create anchored assignments, how they maintain persistence, and how they are actually applied to the layers.

Creation For every layer, a user can add an arbitrary amount of assignments. To add one assignment, we provide a graphical assistant. Here, the user can input the key/value pairs and specify the target properties. Then the user clicks somewhere in the city to specify Pos. Finally, he clicks on a specific element in the layer to set the anchor.

Persistence Every layer has a list of assignments, which are stored separately from the urban layout. This ensures that deletions of elements in the layout do not delete the assignments, this way they are persistent. However, when an element is deleted, we have to update every anchor that references to this element. This is done by modifying the anchor to point to the nearest element of the same type instead.

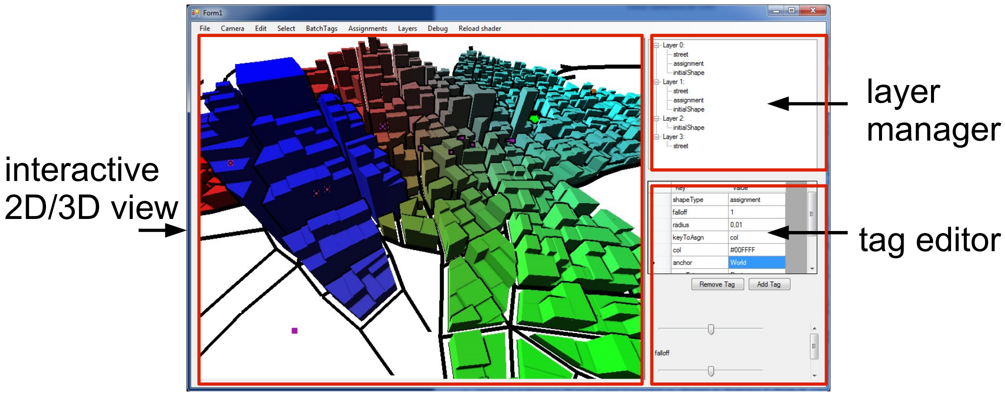
Application The actual application of the assignments to the elements is done during the layer merging process. Recall that two layers are iteratively merged using $L_m = M_f(L_{n-i}, L_m), i = 1 \dots n-1$. Now, every time before M_f is called, we apply the assignments stored for layer L_{n-i} the following way:

For every assignment we do the following: (1) Using the target specification, we search for the elements $\in L_{n-i}$ where the tags should be applied, and add the tags to those elements. In case the target specifies a region with a numerical distribution, the tags are multiplied with those values before they are added (of course multiplication is only performed when the values are of numerical type). (2) In case the same key was already defined in an element, our system keeps the previously specified value. This way assignments on higher levels precede assignments on lower levels.

4.4 Results and Discussion

We have implemented our methods in a stand-alone `C#` application. Urban layouts can be imported from and exported to the CityEngine [Pro10]. To generate final city geometry, the grammar system in the CityEngine is used, based on the tags assigned to objects. In our case, the colors were used to choose different building types. Figure 4.15 shows an example urban layout with geometry created in the CityEngine.

Work flow The user interface allows direct and interactive usage of our introduced methods. When assets are imported, they can be arranged to layers, using a tree-view layer manager. Objects can be freely moved from one layer to another. Visual region-, single- and soft-selection tools using mouse clicks in the interactive viewport are used to define selections. A tag editor allows direct tag assignments, as well as adding anchored assignments. These elements are shown in Figure 4.14. Transformation tools like rotation, scale and translate allow direct and interactive modifications by directly clicking on objects and moving the mouse.



- Layer 0:
  - street
  - assignment
  - initialShape
- Layer 1:
  - street
  - assignment
  - initialShape
- Layer 2:
  - initialShape
- Layer 3:
  - street

Key	Value
shapeType	assignment
falloff	1
radius	0,01
keyToAsgn	col
col	#00FFFF
anchor	World

Figure 4.14: *The user interface of our implementation.*

Artist feedback During the design of our interaction methods, we consulted artists and programmers from a computer game company on their ideas and needs for an urban modeling tool. They noted that the lack of merging of hand-crafted assets as well as the missing direct artistic control was a major disadvantage in previous work. We conducted an informal guided user session. The initial feedback is positive, especially the seamless integration of assets from multiple sources using a layering system and flexible merging is intuitive from an artist's background, and has promising potential to facilitate artist collaboration. They also appreciated the predictability of M_h when merging small city parts.

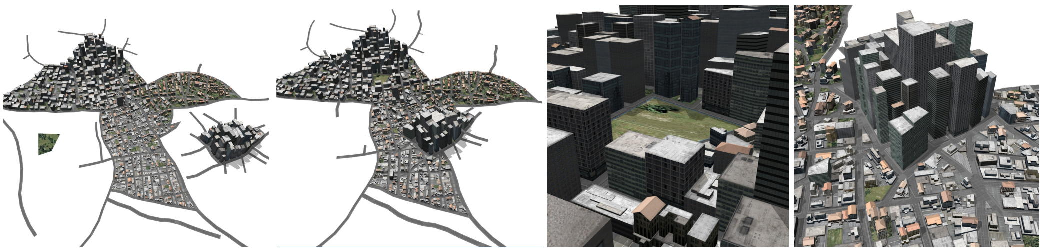


Figure 4.15: *Different editing stages for a city. From left to right: (1) The long street on the left, the park, the main city and the city center on the right are on separate layers. (2) This is the result of moving the layers on top of each other. (3) Close-up of the merged (4) The flexible merging operator was then used to incorporate a part from the street network layout of Rome.*

Performance We achieve interactive frame rates of around 15fps (on an Intel Core2 Quad 6600) for moderately sized cities of about 3,000 parcels and 560 streets on four layers, when moving one layer and performing merging with M_h. Using M_f, the frame rates are around 5fps. In our current implementation, no spatial acceleration structures are employed for intersection calculations, therefore we think that there is a high optimization potential.

Limitations Our current implementation has no mechanism to combine streets or crossings that are relatively near to each other. This can result in very small parcels being generated, which can cause overlaps when actual street geometry is generated. In future work this can probably be solved by introducing a routine to combine nearby objects after merging. Further, the minimal graph cut can sometimes exhibit unintuitive and erratic changes. Especially when the priorities of the two layers are very similarly small, layer movements can cause the minimal cut to jump around a few blocks. This can be observed in the accompanying video, when the layer is moved over an important region in the lower layer. In such a case, the hard merge can be more suitable.

4.5 Discussion

Comparison to previous work Compared to the method of Parish and Müller [PM01], on which the commercial tool CityEngine [Pro10] is based, we contribute persistent and validity preserving transformations of city layouts. Their system does not provide such transformations.

Changing and underlying tensor field [CEW$^+$08], or creating cities using simulations [WMWG09, VABW09] provides *indirect* control over the city layout, in contrast to our approach enabling direct control.

Kelly and McCabe [KM07] proposed a mixture of interactive and procedural techniques: Major roads are created manually, while the minor roads enclosed by main roads are created procedurally. The paper does not describe how to interactively change minor roads or make persistent local changes.

The example based method by introduced by Aliaga et al. [AVB08] stores properties of a street network as parameters of the crossings. A random walk algorithm then connects those crossings and parcels are generated, providing indirect control over the connectivity. In contrast, in our method connectivity is directly specified in the street graph.

The reconfiguration of a city layout defined by a street network with corresponding aerial imagery is explored in [ABVA08]. As aerial imagery is used the main issue here is minimizing the distortions that can occur when transformations are applied to the street network. Therefore a global constraint solver is employed to reduce distortions after a tile is transformed. The main drawback of this method is the global nature of the optimization, as small changes can have effect on the whole city.

Layered procedural modeling Outside the context of city editing, a layer-based procedural editing method that targets single objects and supports triangle meshes was introduced by Schmidt and Singh [SS08]. In contrast to this, we target city-wide modeling and support three semantically different categories (streets, parcels and assignments) for each layer.

Graphcut In image processing, graphcut [FF62] can be used to merge images [KSE$^+$03]. Zhou et. al [ZHW$^+$06] showed how to use graph cuts to merge overlapping mesh regions in the context of geometric texture synthesis. In contrast to our work, they use multiple local graph cuts in overlapping regions, while we use a global graph cut on the whole city layout.

4.6 Conclusion

This chapter presents a city modeling system based on the concept of valid urban layouts. We show that three basic operations, in combination with a layering system, can express all important editing operations on urban layouts. The main advantage of our method is that editing operations like dragging, deletion and insertion, and merging of different layouts from arbitrary sources (procedural or hand-crafted) always produce a valid urban layout. This greatly reduces the cost of editing procedural cities. In the future, we want to add an error function to the flexible merging, with the goal of improving the quality of the graph cut result. Also, we would like to extend the system with convenient tools like merging or snapping to nearby objects.

In addition, elevating complex editing operations to separate layers allows transient operations, like dragging a street over a layout, to only have transient effects on the layout. We have also introduced anchored assignments to allow persistent customizations of a layout in the light of frequent modifications. This enables modifications to an already customized layout (like a finished game level) without destroying the customizations (hand-crafted buildings, game-logic assignments etc.).

Our method combines the power of procedural modeling, which can quickly create large city layouts, with the flexibility of manual modeling, where every aspect of a city can be hand-crafted. Due to persistent assignments, procedural steps and manual assignments can be interleaved and reiterated at any stage of the modeling pipeline.

5

Parallel Generation of Multiple L-Systems

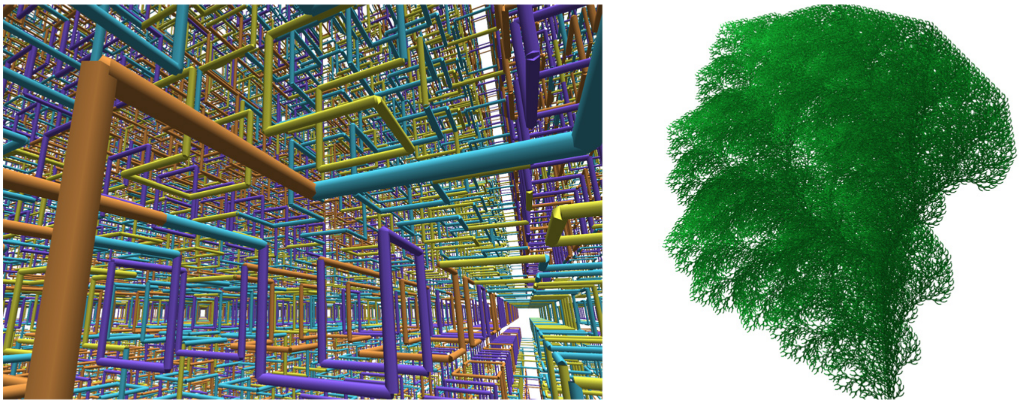


Figure 5.1: *L-systems generated in real-time, at up to 198,000 modules per millisecond: Hilbert 3D space-filling curve and 2D plant.*

In the previous chapters we looked at methods to enable visual and persistent direct control for procedural algorithms. Specific details for performance enhancements were only briefly mentioned. Therefore we will now look at significant performance improvements for specific methods in this chapter. Out of the numerous approaches, we picked L-system based methods. We think this is a first step towards accelerating other methods as well. The results of this chapter have been published under the following reference: [LWW10].

Let us now provide a general motivation for L-system acceleration: Procedural modeling techniques to compute large and detailed 3D models have become very popular in recent years. This leads to the question of how to handle the increasing memory requirements for such models. The current trend is towards data amplification directly on the GPU, for example tesselation of curved surfaces specified by a few control points. This results in low storage costs and allows generating the complex model only when needed (i.e., when it is visible), while also reducing memory transfer overheads. In the same vein, grammars can be viewed not only as a modeling tool, but also as a method for data amplification since a very short grammar description leads to a detailed model.

In this chapter we investigate whether it is possible to efficiently evaluate one of the most classical procedural modeling primitives, L-systems, directly on parallel architectures, exemplified by current GPUs and multi-core CPUs. The main motivation is to enable interactive editing of large L-systems (examples are shown in Figure 5.1) by designers, therefore it is important to speed up the computation of L-systems in order to achieve low response times.

Although L-systems are parallel rewriting systems, derivation through rewriting leads to very uneven workloads. Furthermore, the interpretation of an L-system is an inherently serial process. Thus, L-systems are not straightforwardly amenable to parallel implementation. Previous work has therefore focused on specialized types of L-systems that do not allow side effects in productions, which makes them very similar to scene graphs [LH04]. In contrast, we deal directly with uneven workloads in L-system derivation, and we have identified two main sources of parallelism in the interpretation of L-systems: (1) the *associativity* of traversal in non-branching L-systems, and (2) the *branching structure* itself in branching L-systems.

The main contribution of this chapter is a highly parallel algorithm for L-system evaluation that

- works on arbitrary L-systems , including parametric productions, context sensitive productions, stochastic production selection, and productions with side effects

- works directly on an input string and a plain-text representation of the productions without requiring any compilation or transformation step (e.g., into shaders)

- is efficient in the sense that it requires no explicit inter-thread communication or atomic operations, and is thus completely lock free

- parallelizes both within one L-system as well as among a large number of L-systems

To our knowledge, this is the first L-system algorithm that is highly parallel, i.e. utilizes thousands of threads in an efficient manner. This is achieved by identifying and exploiting the parallelism inherent in L-system derivation using parallel programming primitives like scanning or work-queue management, and a novel algorithm to explicitly resolve the branching structure. We demonstrate that our algorithm outperforms a well optimized single-core CPU implementation on larger L-systems.

Overview An analysis of the intrinsic parallelism of L-systems is provided in Section 5.1. Then our system consisting of two major building blocks will be described: (1) The derivation step will start with the axiom and generate a long string of modules (Section 5.2). (2) The interpretation step takes the string as input and generates the actual geometry (Section 5.3). An extension to support multiple independent L-systems in parallel is shown in Section 5.4.

5.1 Analysis of Parallelism in L-System

5.1.1 Derivation

As an L-system is by definition a module string rewriting system utilizing *parallel* module replacements, the domain of parallelization is obvious: We simply assign chunks of the modules uniformly to multiple threads and perform the rewriting in parallel. The rewritings themselves are independent and thus do not need inter-thread communication. However, the output strings need to be concatenated again, which creates a dependency between the threads. The major problem here is that the length of these strings can vary greatly: for a chunk containing n modules, the minimum expanded module amount is n. This case occurs when no production can be applied and thus every module is copied unmodified to the output. However, the maximum amount of modules is m^n, when the production with the maximum amount m of modules in the successor gets applied to each module.

Therefore, a parallel implementation has to efficiently cope with highly incoherent output module counts for each chunk. Previous shader-based approaches rely on the graphics pipeline to handle concatenation by load balancing (i.e. different output sizes of the geometry shader), which is not ideal because it can lead to serialization, and only works for special types of L-systems. In Section 5.2 we show a native parallel solution to this problem utilizing the scan primitive.

5.1.2 Interpretation

The interpretation of a derived word is defined in a serial manner: Starting with an initial turtle state from the beginning of the module string, the position in the module string is advanced one by one, while applying a modification to the turtle state as defined by the letter of the current module. Therefore, the turtle state of every module string position is dependent on all previous turtle states. While it may look like there is no parallelism to exploit here, there are two inherent parallel concepts that can be extracted, as shown next.

Associative Operations As mentioned before, most turtle commands and the turtle state can be represented as 4x4 matrices, except the push and pop commands. Further, as 4x4 matrix transformations can be combined, we can represent the turtle state up to a specific module string position using one matrix. The key point to parallelize the interpretation is to exploit the *associativity* of those matrix multiplications by accumulating matrices in each parallel chunk locally and combining them in a separate pass using a scan operation, as described in Section 5.3.1.

Inherent Branch Hierarchy Since push and pop commands cannot be represented as matrices, the matrix approach cannot be applied for branching L-systems. Fortunately, the push/pop commands create another type of implicit parallelism that we can exploit: Every time a module representing a push command is encountered, two independent interpretation branches are possible: the module string directly following the push command, and the module string following the corresponding pop command. Thus we can split the work at this point into two threads, as shown in Section 5.3.2.

5.1.3 Multiple L-Systems

Another possibility for parallelism is deriving and interpreting multiple independent L-systems in parallel, for example interpreting multiple trees in a forest. A trivial approach to achieve this is to launch independent threads for every L-system. However, there are two problems with this approach: First, CUDA does not allow running different programs (also called kernels) in parallel. Every thread has to use the same program with possibly different input data. Therefore, in order to interpret different L-Systems in parallel, we need one flexible program that is able to handle all input L-systems. Second, we also want to achieve work-load balancing between the different L-system threads, so there has to be some communication between them. In Section 5.4 we show how to solve those problems in a data-parallel fashion.

5.2 Parallel Derivation

First we show how productions and module strings are efficiently represented on the GPU. Then we introduce the algorithm to perform one iteration of the derivation.

5.2.1 Efficient L-system Representations

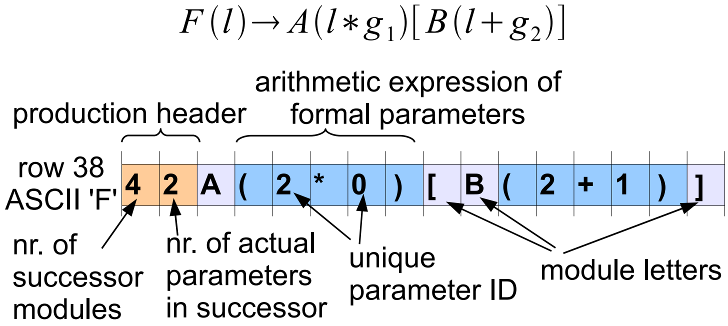


Figure 5.2: *An example production of a parametric L-system packed in a texture.*

In order to allow fast and efficient access to the productions, we store them in a 2D texture in the GPU version. The global parameters are also stored in a texture. For the multi-core CPU version we use a 2D array.

The successor is stored in the row indexed by the ASCII-value of the predecessor's letter. To resolve collisions of two productions that have the same predecessor letter, we create collision chains similar to hash tables.

We perform two simple optimizations: First, we count the number of modules and parameters occurring in the successor for later reference. Those values are stored in a designated header area. Second, in order to allow $O(1)$ parameter value lookup during the derivation, we translate every parameter to a unique numerical ID. To differentiate between local and global parameters, we define that all IDs over a certain threshold l_t identify local variables. This process is visualized for parametric L-systems in Figure 5.2. In order to store stochastic or context sensitive productions, we extend the header area accordingly, by adding the production probability or respectively the left and right context letters.

Representing the Module String A module string contains n module letters. As every module may have an arbitrary amount of parameters assigned, we use an additional array of size n containing indices to an array of actual parameters. One advantage of this separation is that we can simply ignore the parameter array when we have an unparameterized L-system, thus removing the overhead of parameter storage. Figure 5.3 visualizes one module string.

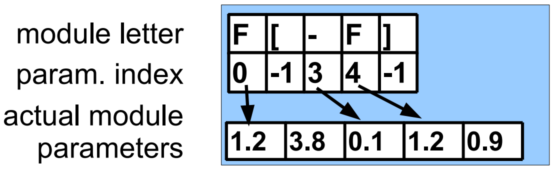


Figure 5.3: *A module string represents a specific state during derivation. We store it as an array of module letters, a parameter index and the actual parameters.*

5.2.2 Derivation

First, we prepare the axiom module string on the CPU side. In the GPU version we then upload it to the GPU. This step is extremely fast, as the axiom usually consists of just a few modules. For the desired iteration amount, we perform one iteration after the other on the GPU or the multi-core CPU. One iteration of the derivation takes the current module string as input and creates an expanded output module string. Between the iterations we swap the pointer to the input module string with the output module string.

The method to compute one iteration in parallel consists of three passes (or “kernels” of the parallel programming language) (see Figure 5.4):

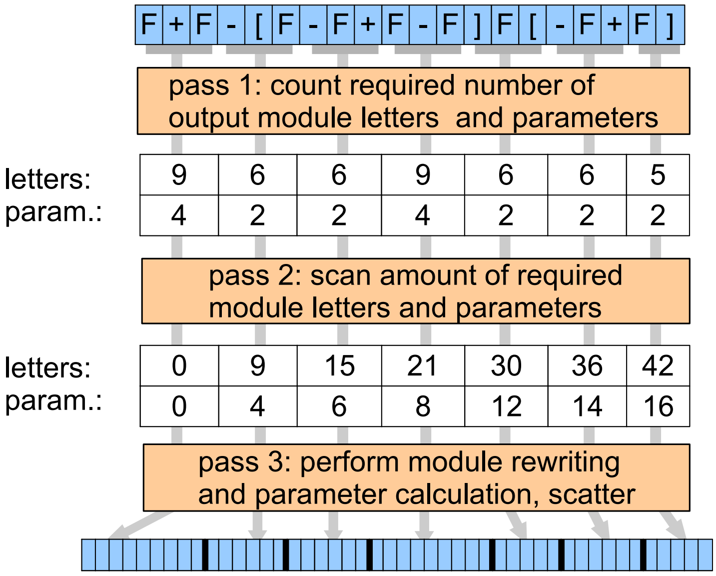


Figure 5.4: *Three passes are performed during each iteration, transforming an input module string to an output module string. For better readability, we show only the letters of the input modules, and omit the parameters.*

1. *Count.* We launch a kernel with n threads. $m = inputSize/n$ subsequent modules from the input module string are assigned to each thread.

Each thread visits all m assigned modules, and fetches the amounts of required output module letters and parameters from the header section of the corresponding production. Those amounts are accumulated for all assigned modules, and finally written to an array in global memory.

2. Scan. We perform a sum-scan operation on this array, in order to calculate offset positions for the scattering of the result.

3. Rewrite. Again, m threads are launched, but this time the module replacement and parameter calculation is actually performed. This is done by fetching for each assigned module the matching production. If a production is *context-sensitive*, we also compare the left and right module letters of a module with the letters stored in the production header in order to determine if the production is applicable. There is no problem in performing a context search across chunk boundaries, as the whole input module string is stored in global GPU memory, and is thus visible to all threads. For *bracketed context sensitive* L-systems the context search is more involved, as we need to take the push and pop commands into account. Therefore, for those systems, we perform a parallel hierarchy extraction step as explained in Section 5.3.2 before we start one iteration. For *stochastic* productions, we determine a random value for every applicable production, using a texture containing random values indexed by the position in the module string. This value is multiplied with the probability stored in the rule header. We then choose the rule with the highest result of the multiplication.

After having decided which production to use, we evaluate the parameters for every module in the production's successor, and insert the resulting successor modules into the result module string. The parameter evaluation based on the arithmetic expression of formal parameters is conducted by a simple mathematical expression parser in the kernel. When no production is defined for a module, we simply copy it unmodified to the output. As we have the offset values to index the module string, every thread can write its resulting modules without interference from the other threads.

Alternative Method Our algorithm requires three passes for each iteration. An alternative approach is to implement the module string as a linked-list of modules, which is modified with atomic operations. This requires only one pass. The amount of atomic operations can be reduced by using batched linked lists, where each element contains multiple modules. However, when we implemented this alternative approach it turned out to be considerably slower than the three-pass approach, probably caused by the implicit serializations occurring on concurrent atomic operations.

Possible Enhancement Our algorithm does not use shared memory between threads on the same multiprocessor for communication. Maybe applying the shared-memory aware compaction model presented by Billeter et al. [BOA09] could further improve performance.

5.3 Parallel Interpretation

The result of a derivation is a module string. This needs to be converted into a geometric representation. There are two cases allowing two different parallel algorithms: non-branching and branching L-systems.

5.3.1 Non-Branching Module Strings

As explained in Section 5.1, most modules can be represented as associative matrix transformations. We can exploit this efficiently to interpret non-branching L-systems by interpreting chunks independently. We present a three pass algorithm (see Figure 5.5):

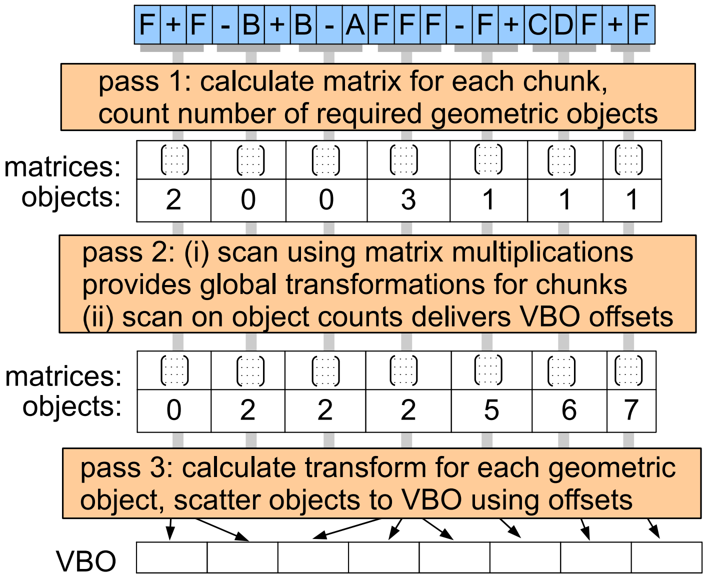


Figure 5.5: *By exploiting the associativity of modules representable as matrix operations, we can efficiently interpret non-branching L-systems with this algorithm.*

1. Matrix accumulation The string is split into m chunks, each chunk is assigned to an independent thread. In each chunk, we combine the matrices corresponding to the modules in the chunk, resulting in one local transformation matrix. Further, we count the amount of geometry generated in the modules. Both values are stored in an global array.

2. Matrix scan. A parallel scan operation is performed on the matrices, using 4x4 matrix multiplication as the operator. The resulting array contains matrices representing a global transformation of the turtle state to the start of each chunk. Additionally, a scan using integer additions on the objects array is performed to calculate offsets for the geometry to be created.

3. Geometry generation. Finally, to get the global positions of each geometry object, we again operate on m chunks in parallel as in 1. and accumulate module matrices. But this time we do not start with the identity matrix but with the matrix determined during the scan. Furthermore, every time we encounter a geometry generation module, we calculate the global position of the object and insert it into a vertex buffer object (VBO), using the offsets determined during the object scan.

The idea to use a scan to determine VBO offsets for tessellating parametric surfaces was introduced by Schwarz and Stamminger [SS09].

5.3.2 Branching Module Strings

For branching L-systems, parallelization is achieved by exploiting the branch hierarchy. Whenever a push command opening a new branch is encountered, two independent new work items are generated: one for the branch and one for the remaining string following the corresponding pop command. The main problem is to find the pop command in an efficient (i.e., parallel) way. This information is also necessary for fast context search in bracketed context sensitive systems. We therefore first present a novel parallel algorithm to extract the hierarchy, and then show how the work items can be efficiently managed.

Parallel Hierarchy Extraction

One critical observation is that when looking only at a particular hierarchy depth in the branch hierarchy, corresponding push/pop pairs follow each other directly. The main idea is therefore to extract the push and pop commands from the module string and sort their positions into buckets according to their depths. Each bucket will then contain the positions of corresponding push/pop pairs. These can then be easily traversed to store with each push the position of the corresponding pop.

We introduce an efficient parallel algorithm based on this idea that does not require direct communication between the blocks. We assume that we know the maximum depth of d_{max} of the branching hierarchy and allocate a two-dimensional bucket sort array with d_{max} rows. The complete algorithm is visualized in Figure 5.6, and consists of 5 passes operating on uniform chunks in parallel:

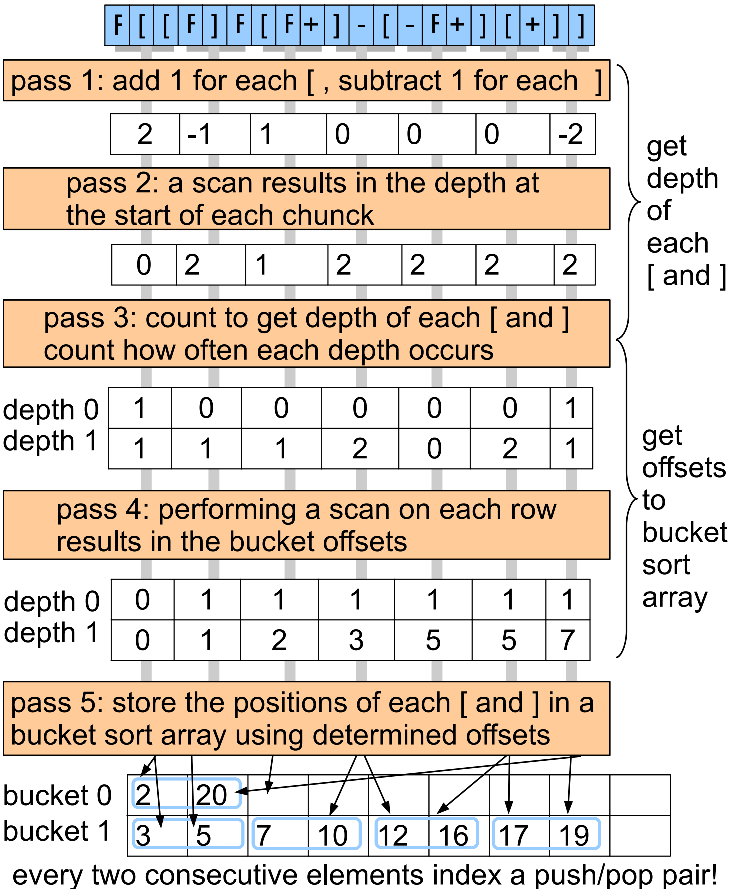


Figure 5.6: *This algorithm allows efficient and parallel searching for corresponding push and pop pairs.*

1. Chunk depth calculation. Starting at zero, we add 1 for every push, and subtract 1 for every pop occurring in a chunk. This results in the depth of the chunk end relative to the chunk start.

2. Depth scan. Performing a scan of those values results in the absolute depths of the start of each chunk.

3. Depth-based push/pop count. Now, by starting at the calculated absolute depth of the chunk start, we can determine the absolute depth of every push and pop occurring in the chunk. We use this to to determine the offsets for the bucket sort array by counting the amount of push and pop commands m_i in each depth j in the following way: $c_j = \sum m_i \mid d_i = j$. We store the values for $c_j \mid 0 < j < d_{max}$ in a global array.

4. Scan push/pop counts. The scan of the c_j arrays results in the bucket offsets o_j each thread has to use in order to allow conflict-free writing to buckets.

5. Write push/pop locations. Again, we visit every push and pop command m_i of every chunk. But this time we write the absolute input module string position of the module m_i in the bucket d_i using the offset positions determined previously. This ultimately leads to a bucket sort array where every two consecutive elements in a row correspond to a push and pop pair.

Analogously to the matrix interpretation algorithm, we also calculate VBO offsets needed to scatter the geometry. In our implementation, this process is combined with pass 1, and the VBO offsets are stored as a parameter of push commands.

Memory Footprint In order to reduce the memory footprint of the bucket sort array, we actually use a one-dimensional array instead of two dimensions. This allows us to pack the bucket arrays for the individual depths without empty spaces tightly together. In the worst case, when every module in the module string is a push or pop module, the number of required memory elements is then equal to the amount of modules. The offsets needed for the 2D to 1D packing can simply be calculated form the values obtained in iteration 4: $c_j + o_j$ of the last chunk equals the total amount of elements in a specific bucket. When we perform a scan operation of those values for each bucket we get the offsets to map the 2D bucket arrays into a 1D array.

Integration into Module String As a last step, we use the information bucket arrays to write the position of corresponding push and pop modules directly into the module string to allow fast access during interpretation. This is a simple parallel algorithm: We evenly assign the 1D bucket array to multiple threads. Every even element in this array contains the position of a push command, every odd element references a pop command. Thus we need to write the position stored at every odd element as a parameter to the push module referenced by the preceding even element.

Work Queue-based Interpretation

As a result of the previously explained algorithm, every push module has a parameter indexing the position of the corresponding pop module, as well as a VBO offset parameter. With this information, we use a parallel work-queue approach [ZHWG08, LGS$^+$09, PO08]:

First, we define a work item as a tuple (M, i) where M is a 4x4 matrix representing the turtle state, and i is the array index of a module in the module string. The interpretation is started with the work-queue item $(I, 0)$, where I represents the identity matrix. Then, one thread starts serially interpreting the module string using this work item. When a push-module is encountered, the thread creates two work items (M, i_1) and (M, i_2), where M is the current turtle state, i_1 is the current module index and i_2 is the index of the corresponding pop command. Then it puts (M, i_1) on a thread local work stack and continues with (M, i_2). Directly continuing with (M, i_2) is important for good performance, as it induces a breadth-first traversal (with respect to the branch hierarchy) which enables faster spreading of the work items to multiple threads.

To actually distribute the work items between threads, we use parallel work queue management [ZHWG08] [LGS+09] [PO08] in the following adapted way: When the local work stack of a thread is full, we write the contained items to a work item array in global memory. Each thread is assigned a unique offset to this array. When all threads are finished, we create a compact array of indices to the work item array using scan operations. These indices are then evenly distributed to threads, which execute the tasks as described previously. The process is iterated until no more work items are left.

Alternative approach In our approach each thread has an independent local work item queue. During kernel execution, no intra-block distribution of work is performed using the shared memory. We have also implemented and tested a version utilizing shared block local memory to distribute threads. However, this method turned out to be slower, probably the overhead of the multiple required intra-block synchronization points is higher than the possible gain achieved through faster work distribution.

5.4 Multiple L-systems

The algorithms presented in the previous sections always work on one L-system at a time. However, it would be beneficial to derive multiple L-systems with different parameter sets in parallel at the same time. An example application for this is designing procedural forests using L-systems, as shown in Figure 5.7. Therefore we introduce extensions to the previously introduced algorithms, allowing the derivation of multiple L-systems in parallel while still retaining work-load balancing between threads.

5.4.1 Representation of Multiple L-systems

For every L-system L_i with the unique identifier i, we need to define a unique set of productions, global parameters and an initial turtle state T_i. In Section 5.2.1 we showed how one L-system production and global parameter set is stored in a texture. In order to store multiple L-systems, a unique offset O_i into this texture is assigned to every L-system. The offsets can be trivially calculated by summing the amounts of productions and parameters of every L-system, and ensuring that the values stored in the texture do not overlap.

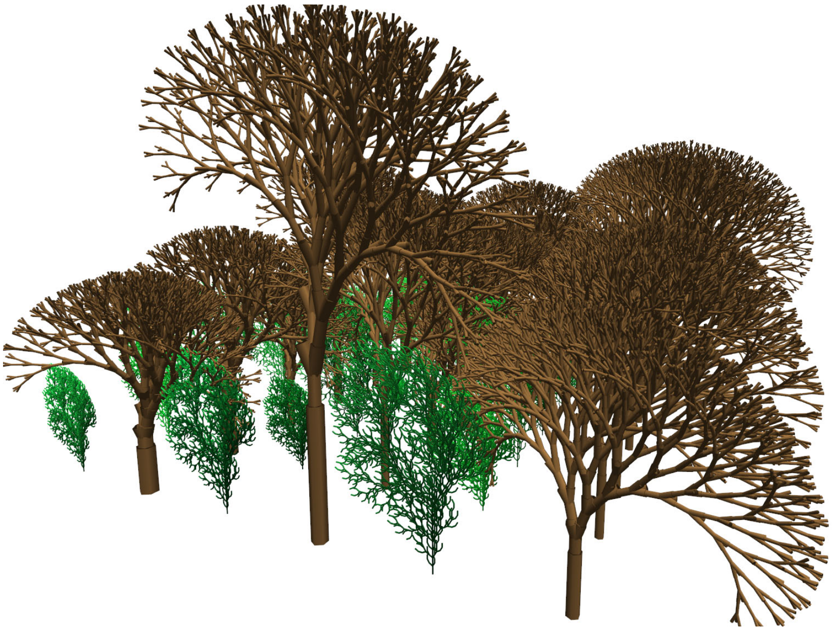


Figure 5.7: *Here multiple different L-systems with unique parameters are shown. The derivation and interpretation can still be performed interactively.*

In order to access this representation during derivation, a lookup table from L_i to O_i and T_i is created and stored in global memory.

5.4.2 Derivation of Multiple L-systems

Before the derivation algorithm is started, we connect the axioms of every L-system to one combined axiom. At the start of every axiom a delimiter symbol (we have arbitrarily chosen 'M') is inserted. In order to keep track of which letter belongs to which L-system during derivation, the following algorithm is performed before every derivation iteration: First we assign the letters uniformly to threads. Then every thread counts the amount of delimiter symbols in its assigned chunk. In a second pass, an additive scan is performed on those values, resulting in the L-system index of the first letter for each chunk. This algorithm is outlined in Figure 5.8.

Afterwards the derivation is performed. Every time a rule or parameter needs to be fetched, we determine the texture offset from the look up table using the L-system index. When a delimiter symbol is encountered, the index is increased by one. In the last pass of the derivation, we also write the positions of the delimiter symbols in the module string to a global array.

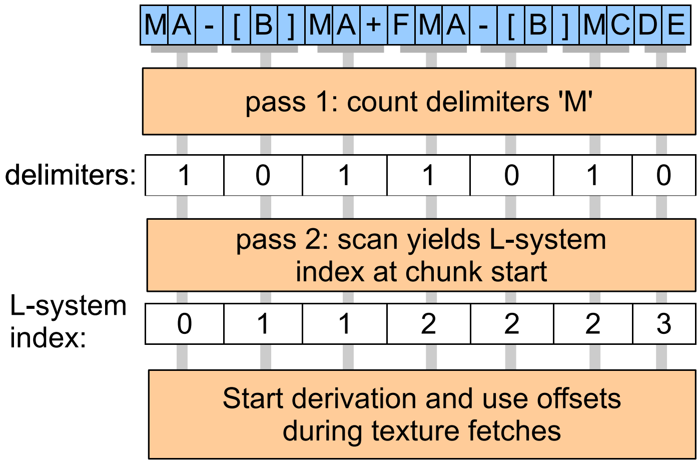


Figure 5.8: *In order to determine which L-system a letter belongs to, this algorithm is performed before every derivation iteration.*

Different Iteration Amounts In order to support different numbers of iterations for the L-systems, the number of iterations to be performed is also stored in the look up table. This value is compared with the current iteration number during the derivation. When it is higher, no rule replacement is performed for this L-system.

5.4.3 Interpretation of Multiple L-systems

For non-branching L-systems, the algorithm in Section 5.3.1 needs to be modified. Instead of a scan of all matrices, a segmented scan of the matrices needs to be performed. The segments are defined by the delimiter symbols. The matrix of the first letter after a delimiter symbol is set to the initial turtle state stored in the lookup table.

As we know the string positions of each delimiter symbol after the derivation, the adaptation of the branching interpretation algorithm described in Section 5.3.2 is straightforward. We simply create a work item for every delimiter symbol. In every work item the L-system index is stored, and the turtle state is set to the one in the look up table. Then the interpretation algorithm is performed as described in Section 5.3.2.

5.5 Results

We implemented our parallel algorithms for GPUs utilizing CUDA and for multi-core CPUs using `POSIX` threads. In CUDA, up to 1920 threads are utilized (60 blocks of 32 threads), the multi-core CPU version uses 4 threads. We compare those implementations against a highly optimized single-core CPU version, created with the help of performance profilers to detect and remove bottlenecks. This version has the advantage that no kernel or thread launch overheads occur, and that no scan or multi-pass operations are necessary. The main low-level optimizations include the complete avoidance of advanced C++ features like virtual functions and dynamic memory allocations during runtime. Those optimizations resulted in a speedup of multiple orders of magnitude compared to our initial single-core CPU implementation. The test platform was an Intel Core 2 Quad Q6600 2.4GHZ PC with a Geforce GTX 280 graphics card.

L-system	bracketed	parametric	stochastic	context sens.
Hilbert 3D, pg. 20				
Koch curve, pg. 10 (d)				
row of trees, pg. 48		✓		
2D plant, pg. 25 (c)	✓			
3D tree, pg. 60 (b)	✓	✓		
plant stochastic, pg. 28	✓		✓	
p. ctx. sens., pg. 35 (b)	✓			✓

Table 5.1: *Property matrix of the L-systems shown in our results. The page numbers refer to the L-system definitions by Prusinkiewicz and Lindenmayer [PL96].*

Test Scenes We used seven test scenes to demonstrate several aspects of our system. In order to ensure repeatability and comparability of our results, all our L-system productions are directly taken from Prusinkiewicz and Lindenmayer [PL96] for our performance measurements. In Table 5.1 we classify the test scenes according to the properties of the used production set.

Rendering Our implementation creates a VBO containing lines. For our figures we use a geometry shader during rendering, creating cylinders from the lines. All our performance measurements do *not* contain the rendering times, as the rendering times are the same both for the CUDA and the CPU versions. Neither do the measurements contain the CPU-GPU memory transfer times required by the CPU versions, which we measured in the range from 20–40ms, making the CPU versions very hard to use in a real-time rendering setting.

		total derivation times			
		ms	rel. speedup		
L-system, i	modules	1 core	4 cores	CUDA	d_f
Hilbert 3D, 6	1,266,864	4.70	3.76×	2.92×	5
Koch curve, 7	915,049	3.45	3.26×	3.20×	6
row of trees, 10	815,545	10.21	3.29×	3.15×	8
2D plant, 7	813,169	3.31	3.04×	3.15×	6
3D tree, 16	622,334	8.53	1.40×	1.17×	13
plant stoch., 11	835,481	6.45	1.75×	3.23×	9
p. ctx. sens., 30	25,174	0.73	0.39×	0.03×	?
multi L-sys., 10	2,751,022	30.51	1.48×	1.31×	4

Table 5.2: *Derivation performance measurements. i shows the amount of iterations performed. The single-core CPU times are absolute values in milliseconds, the multi-core CPU and CUDA values are relative speedups compared to the single-core CPU values. d_f denotes the first iteration where CUDA is faster compared to the single-core CPU version during derivation.*

Scalability We evaluated how our derivation and interpretation scale with the number of iterations. For all our test scenes, we measured how long one specific iteration i of the derivation takes, and calculated the number of modules generated per microsecond during each iteration i. The results for three L-systems are shown in Figure 5.9. For the interpretation, a specific amount of derivation iterations was performed prior to the interpretation, the interpretation time was measured, and the amount of modules interpreted per microsecond was calculated. The results are shown in Figure 5.10. For readability only three L-systems are shown, but all results show a similar pattern: As expected the initial iterations incur some overhead in the parallel implementation on the GPU and the multi-core CPU, because the amount of parallelism is limited, and the overhead of launching CUDA-kernels or `POSIX` threads is a significant factor. This makes parallel versions slower on the first few iterations. For the later iterations the parallel implementations are several times faster, because a high amount of threads can be utilized. For all L-systems, we list the first derivation iteration d_f where CUDA is faster compared to the single-core CPU version, as well as the first interpretation of a string generated with i_f iterations where CUDA is faster in Tables 5.2 and 5.3. The total performance including the cases where CUDA is slower will be discussed in the next two paragraphs.

Total Derivation Performance The CUDA and the multi-core CPU version are very similar in performance and are significantly faster than the single-core CPU version in most cases. There are two notable exceptions: First,

	total interpretation times				total speedup	
	ms	rel. speedup			deriv.+interpr.	
L-system, i	1 core	4 cores	CUDA	i_f	4 cores	CUDA
Hilbert 3D, 6	31.50	1.23×	6.61×	5	1.35×	5.68×
Koch curve, 7	22.56	0.70×	2.99×	6	0.78×	3.02×
row of trees, 10	77.04	4.25×	10.84×	7	4.11×	8.43×
2D plant, 7	22.45	1.27×	1.21×	7	1.37×	1.31×
3D tree, 16	31.44	2.78×	3.87×	13	2.30×	2.59×
plant stoch., 11	14.70	0.09×	0.24×	?	0.13×	0.33×
p. ctx. sens., 30	0.11	0.43×	0.11×	?	0.39×	0.03×
multi L-sys., 10	29.12	1.65×	2.22×	4	1.56×	1.64×

Table 5.3: *Interpretation performance measurements.* i_f *denotes the first iteration where CUDA is faster compared to the single-core CPU version during interpretation. The total speedup is shown in the last two colums.*

the 3D tree is only marginally faster. Second, the context sensitive plant is considerably slower. Both cases can be attributed to the following observation: The corresponding L-systems are growing rather slowly, compared to the other test cases. For the 3D tree 622,334 modules are created in 16 iterations, while the plant creates only 25,174 modules in 30 iterations. All other tested L-systems create more modules with a lower iteration count. Therefore the other L-systems have less relative thread launch overhead. To sum it up, during the derivation the parallel implementations are significantly faster when the L-systems grow relatively fast. Another important thing to note is that the results for the stochastic system vary with the random seed, our measurements were in a range of about +/- 20% for different seeds.

Total Interpretation Performance The three tested non-bracketed (serial) L-systems are significantly faster in CUDA compared to the other versions, probably because the parallel matrix interpretation makes good use of the high arithmetic density of the GPU. On the other hand, the multi-core CPU version performs rather bad on those L-systems (with the exception of the row of trees L-system), probably the matrix multiplications and the memory bandwidth are the limiting factors here.

The results for the branching L-systems vary. The first thing to note is that the five-pass hierarchy extraction step requires considerably less time than the actual interpretation. For reference, a hierarchy extraction takes 3.3ms on one CPU core and 1.97ms in CUDA for the 2D plant. Our interpretation of the varying results is that the L-systems have different branching structures, which directly affect how effective our work-queue interpretation is: The 3D tree (Figure 5.11) has very regular branching, and is considerably faster to

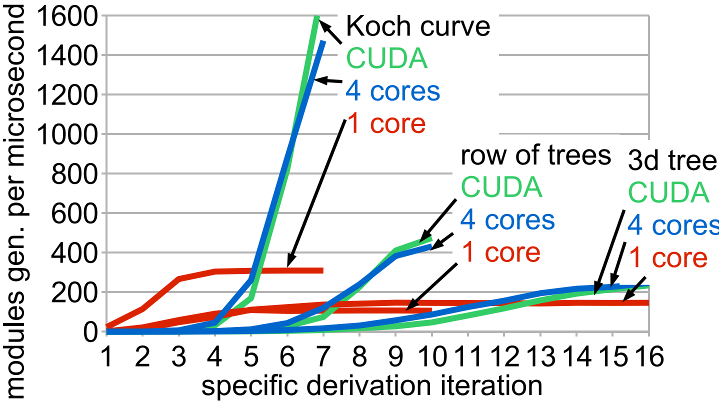


Figure 5.9: *Scalability analysis of the derivation step. For every iteration, we calculate the number of modules generated per microsecond.*

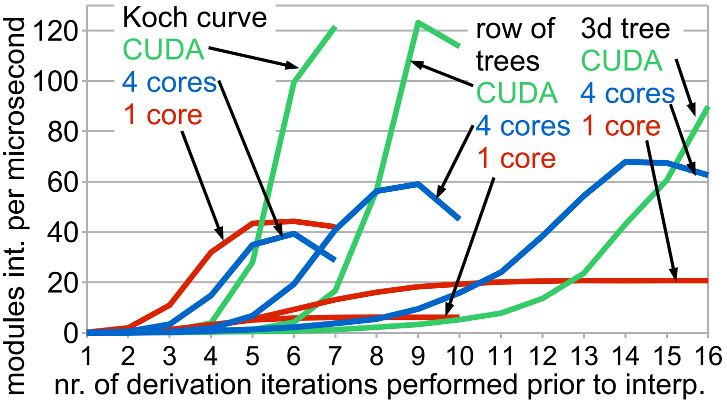


Figure 5.10: *Scalabilty analysis of the interpretation step. We performed a specific amount of derivation steps before the interpretation was performed.*

interpret with the parallel versions, while the 2D plant (Figure 5.1) exhibits more irregular branching, resulting in only a small speedup. The stochastic plant contains only a few long branches with many small ones attached (Figure 5.11), making it hard to spread the work to multiple threads. The context sensitive plant is even harder for the parallel algorithms to interpret, as the amount of modules is very low compared to the other cases. In summary, the non-bracketed L-systems are considerably faster in CUDA, while the bracketed L-systems create varied results based on the branching structure.

Total Speedups The combined derivation and interpretation speedups are shown in the last two columns of Table 5.3. In all but the context sensitive and multiple L-system case, the interpretation time is much higher than the derivation time, therefore the combined speedups are mainly dependent on

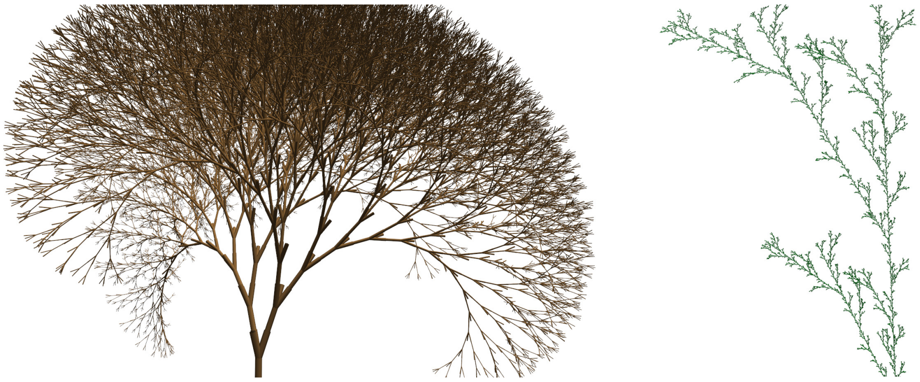


Figure 5.11: *L-systems generated in real-time: 3D tree and stochastic plant.*

the interpretation speedups.

Multiple L-Systems The performance of multiple L-systems is tested using a scene consisting of 38 2D plants with 4 iterations and 12 3D trees with 10 iterations, similar as shown in Figure 5.7. The results are shown in the last row of Tables 5.2 and 5.3. While the multi thread versions consistently show a speedup of up to 2.22 times, it is not as high as we had expected. It is important to note that we did not optimize the multiple L-system implementation, while the single-core version is highly optimized. We believe that there is still room for improvement.

5.6 Discussion

Comparison to Previous Work The main advantage over the previous GPU-based methods [LH04, Mag09] is that we make explicit use of parallel primitives and do not rely on the graphics pipeline to deal with data amplification and other issues. We fully support productions having side-effects and thus do not need to rely on the specific side effect-free turtle commands presented by Lacz and Hart [LH04]. Furthermore, we can directly use the productions without requiring a compilation or transformation step. Compared to the multi-CPU based method proposed by Yang et al. [YHL$^+$07] our algorithm does not need an intermediate scene-graph representation of the module string. Furthermore our algorithm can utilize thousands of threads, which is significantly higher than what was shown in the multi-CPU version.

Memory Transfer to Graphics Hardware One important advantage of our CUDA version is that the resulting geometry already resides in GPU memory, so there is no need for a copy operation. The CPU versions, on the other hand, needs to perform a copy from the main system memory to the

GPU. We measured copy times of about 20–40ms for the tested L-systems – this is very high compared to the generation times, increasing the total speedup of CUDA significantly, and showing that a GPU implementation is highly desirable. All our results do *not* include those transfer times.

Intra-Block Thread Divergence In CUDA, when different execution paths are taken within a sub-block of threads (called warp), those execution paths are serialized, reducing the utilization of the intrinsic SIMD capabilities. In our algorithm, the following situations lead to divergence in the code: (1) If two productions produce a successor of different length during derivation. This divergence is directly caused by the varying data amplification, and can probably not be avoided. (2) During interpretation, the matrix notation helps in maintaining thread coherence, as we can perform the multiplications coherently after each thread decides which matrix to use based on the associated commands. However, when some threads either have no command to perform or have a geometry generation command, SIMD can not be fully utilized. (3) In the work-queue algorithm, the number of elements in a local work queue can vary, leading to divergence. As pointed out, in an alternative approach we removed this divergence by enabling intra-block work item sharing, which turned out to be slower than having divergence. (4) The length of one work item can vary, leading to divergence. Unless we would further split work items into sub work items, we can probably not avoid this divergence.

Limitations The varying results of the work-queue approach indicate that there may be future work necessary in creating more consistent speedups, maybe a more elaborate work-queue management can achieve this. As for the tested context sensitive L-system, the high iteration counts result in a low performance of the CUDA approach, making the use of the CPU version more appropriate in this case. Stochastic productions can be used to simulate biological plant-internal signals [PL96], maybe there is a way to abstract this simulation that is more amenable to parallel interpretation.

5.7 Conclusion

In this chapter we introduced a solution to generate L-systems on a parallel architecture. We make two major contributions. First, we show how parallel primitives can be employed to handle the varying data amplification during derivation. Second, we introduce an algorithm to match the push and pop stack operations to obtain a parallel implementation of L-system interpretation. The system can work with a broad set of rules, including parametric rules, stochastic rules, and context-sensitive rules. It can parallelize a single L-system as well as multiple independent L-systems. We have demonstrated that our parallel L-system outperforms a highly optimized single-core CPU implementation in many test cases, while there are some cases where the single-core version is faster. The advantage of our GPU version gets more pronounced when taking into account CPU-GPU memory transfer times required by the CPU versions.

Future Work We would like to integrate the parallel derivation of L-systems in a rendering engine to render large-scale environments. We plan to combine the derivation of L-systems with occlusion queries and memory management algorithms so that we can render environments several times the size of graphics card memory in real time. Also, it would be interesting to extend the work to procedurally generated architecture, and more complex L-system concepts.

6

Summary

The goal of this thesis was to bring direct artist control to procedural methods, show novel visual interaction techniques enabled using such methods, as well as to improve the performance of procedural methods to real-time speeds.

To this end, we introduced new paradigms and algorithms for direct and visual artist control of procedural content generation. We presented a visual grammar editor for split grammars and visual high-level tools to modify city layouts. Artists of a game company employed our software prototype and provided feedback. Additionally, a novel algorithm for parallel generation of grammars was introduced. Let us focus on those contributions in greater detail:

Artist control Two novel paradigms for persistent direct control were introduced. For architecture, we enable direct control by introducing semantic tags to the rules. Direct visual editing operators like drag and drop are automatically mapped to locators and changes to steer the derivation after global changes. An abstract view of this paradigm is shown in Figure 6.1.

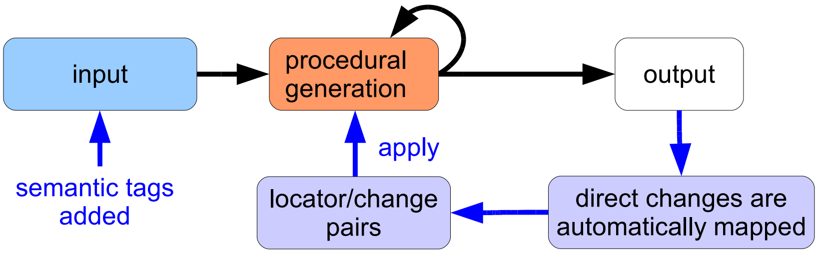


Figure 6.1: *Abstract view of our first novel persistence paradigm: Semantic tags are (automatically) added to the input. Direct changes are automatically mapped to locator/change pairs. Those changes are applied during procedural generation.*

For cities, we introduced a layering system, allowing to merge content generated from various procedural algorithms. Persistent direct changes are supported by moving changed elements to a new layer, and merging them to the final result after a global change. The main advantage of this approach is that the procedural generation algorithm does not need to know anything about the layering system, thus essentially we decouple procedural generation from persistent direct control. This is visualized in Figure 6.2.

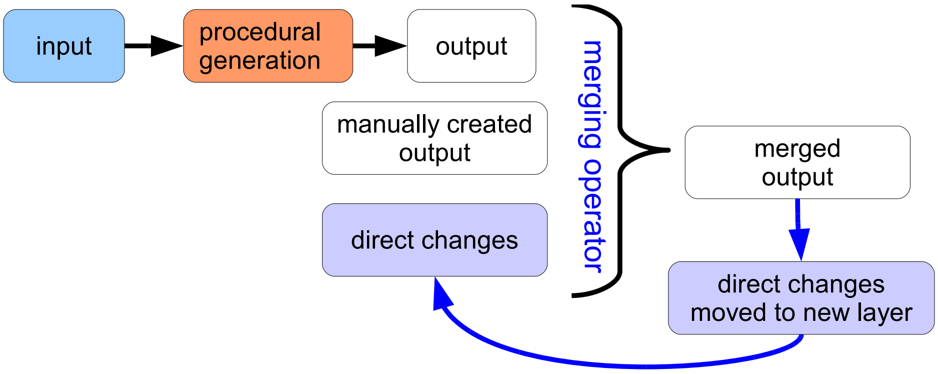


Figure 6.2: *Abstract view of our second novel persistence paradigm: Content from arbitrary sources, manually created or automatically generated, is merged into one output. Direct changes to this output can be moved into a new layer, making them persistent even when one of the inputs changes.*

Novel interaction techniques We showed that using our visual grammar editor, building grammars can be constructed in about ten minutes. Visual high-level operations enable intuitive direct control without modifying the grammar. A layering system for cities allows artists to collaborate in a direct way, and high-level visual operations can be performed without manual remodeling of city parts.

Real-time performance In Chapters 3 and 4 we provided some implementation details to achieve real-time speeds of the corresponding algorithms, as well as brief performance evaluations showing real-time speeds are obtained.

In Chapter 5 this was done in much greater detail, by introducing and evaluating a novel algorithm for L-system generation, based on the parallel scan primitive, which is able to spread the work to thousands of threads. We see this as a first step towards accelerating other grammar-based approaches as well.

Expected benefits The main benefit of our direct control methods is the possibility of significant cost reduction when creating virtual urban environments. Large pars of the city can be created rapidly using procedural methods, while fine-tuning of individual parts is always supported, without the need of laborious adoptions of city parts after minor modifications. This enhances the appeal of procedural methods in general, preparing a wide-spread adoption of such methods in various fields.

6.1 Research Outlook

We have shown that our novel persistence algorithms work for architecture and buildings. It would be interesting to add those concepts to plants or procedural textures. Another interesting future work is extending the parallel L-system algorithm to split grammars, creating buildings. Further, there is currently no method to automatically map direct output changes to input changes, for example changing a rule-base when the output is modified.

List of Figures

List of Tables

Bibliography

[ABVA08] D.G. Aliaga, B. Beneš, C.A. Vanegas, and N. Andrysco. Interactive reconfiguration of urban layouts. *IEEE Comput. Graph. Appl.*, 28(3):38–47, 2008.

[AVB08] D.G. Aliaga, C.A. Vanegas, and B. Beneš. Interactive example-based urban layout synthesis. In *SIGGRAPH Asia '08*, pages 1–10, New York, NY, USA, 2008. ACM.

[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: Proc. of the conference on Virtual reality, archeology, and cultural heritage*, pages 187–196, NY, USA, 2001. ACM Press.

[BFH05] R. Berndt, D. Fellner, and S. Havemann. Generative 3d models: a key to more information within less bandwidth at higher quality. In *Web3D '05: Proceedings of the tenth international conference on 3D Web technology*, pages 111–121, New York, NY, USA, 2005. ACM.

[BOA09] M. Billeter, O. Olsson, and U. Assarsson. Efficient stream compaction on wide simd many-core architectures. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 159–166, NY, USA, 2009. ACM.

[BPF+03] F. Boudon, P. Prusinkiewicz, P. Federl, C. Godin, and R. Karwowski. Interactive design of bonsai tree models. In *CG Forum: Proc. of Eurographics*, volume 22, pages 591–599. EG, 2003.

[BW05] X. Baele and N. Warzee. Real time l-system generated trees based on modern graphics hardware. In *SMI '05: Proceedings of the International Conference on Shape Modeling and Applications 2005*, pages 186–195, Washington, DC, USA, 2005. IEEE Computer Society.

[Cac09] Fernando Cacciola. 2D straight skeleton and polygon offsetting. In *CGAL User and Ref. Manual*. CGAL Editorial Board, 3.5 edition, 2009.

[CEW+08] G. Chen, G. Esch, P. Wonka, P. Müller, and E. Zhang. Interactive procedural street modeling. *ACM Trans. Graph.*, 27(3), 2008.

[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.

[COR07] NVIDIA CORPORATION. Cuda: Compute unified device architecture. http://developer.nvidia.com/. 2007.

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

[EK72] J. Edmonds and R.M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

[FF62] L. Ford and D. Fulkerson. *Flows in Networks.* Princeton University Press, 1962.

[GK07] B. Ganster and R. Klein. An integrated framework for procedural modeling. In Mateu Sbert, editor, *SCCG '07*, 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: Proc. of the 4th international conference on CG, virtual reality, visualisation and interaction in Africa*, pages 161–169, New York, NY, USA, 2006. ACM Press.

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

[HBW06] E. Hahn, P. Bose, and A. Whitehead. Persistent realtime building interior generation. In *sandbox '06: Proc. of the ACM SIGGRAPH symposium on Videogames*, pages 179–186, NY, USA, 2006. ACM.

[HF06] Kai Hormann and Michael S. Floater. Mean value coordinates for arbitrary planar polygons. *ACM Trans. Graph.*, 25(4):1424–1441, 2006.

[HJA02] C. Hoffmann and R. Joan-Arinyo. *Handbook of Computer Aided Geometric Design*, chapter 21: Parametric modeling, pages 519–541. Elsevier, 2002.

[KM07] G. Kelly and H. McCabe. Citygen: An interactive system for procedural city generation. In *Fifth International Conference on Game Design and Technology*, pages 8–16, 2007.

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

[KSE$^+$03] V. Kwatra, A. Schödl, I.A. Essa, G. Turk, and A.F. Bobick. Graphcut textures: image and video synthesis using graph cuts. *ACM Trans. Graph.*, 22(3):277–286, 2003.

[LD99] B. Lintermann and O. Deussen. Interactive modeling of plants. *IEEE CG Appl.*, 19(1):56–65, 1999.

[LGS$^+$09] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. *Computer Graphics Forum*, 28(2):375–384, 2009.

[LH04] P. Lacz and J.C. Hart. Procedural geometry synthesis on the gpu. In *Workshop on General Purpose Computing on Graphics Processors*, pages 23–23, NY, USA, 2004. ACM.

[LWW08] M. Lipp, P. Wonka, and M. Wimmer. Interactive visual editing of grammars for procedural architecture. *ACM Trans. Graph*, 27(3):102:1–10, 2008. Article No. 102.

[LWW10] M. Lipp, P. Wonka, and M. Wimmer. Parallel generation of multiple l-systems. *Computers & Graphics*, 34(5):585 – 593, 2010.

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

[Mag09] M. Magdics. Real-time generation of l-system scene models for rendering and interaction. In *Spring Conf. on Computer Graphics*, pages 77–84. Comenius Univ., 2009.

[MBP05] F. Milicchio, C. Bertoli, and A. Paoluzzi. A visual approach to geometric programming. *Computer-Aided Design and Applications*, 2(1-4):411–421, 2005.

[MK10] M. Magdics and G. Klár. *Rule-based Geometry Synthesis in Real-time*, pages 41–66. A K Peters, 2010.

[MP96] Radomír Měch and Przemyslaw Prusinkiewicz. Visual models of plants interacting with their environment. In Holly Rushmeier, editor, *Proc. of ACM SIGGRAPH 96*, pages 397–410. ACM Press, August 1996.

[MWH$^+$06] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural Modeling of Buildings. In *Proc. of ACM SIGGRAPH 2006 / ACM Trans. Graph*, 2006.

[MWR$^+$09] P. Musialski, P. Wonka, M. Recheis, S. Maierhofer, and W. Purgathofer. Symmetry-based facade repair. In *Proceedings of the Vision, Modeling, and Visualization Workshop 2009*, pages 3–10, 2009.

[MZWG07] P. Müller, G. Zeng, P. Wonka, and Luc Van Gool. Image-based procedural modeling of facades. *ACM Trans. Graph.*, 24(3):85, 2007.

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

[PM01] Y.I.H. Parish and P. Müller. Procedural modeling of cities. In Eugene Fiume, editor, *Proc. of ACM SIGGRAPH 2001*, pages 301–308. ACM Press, 2001.

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

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

[PO08] A. Patney and J.D. Owens. Real-time reyes-style adaptive surface subdivision. *ACM Trans. Graph.*, 27(5):1–8, 2008.

[PPV95] A. Paoluzzi, V. Pascucci, and M. Vicentino. Geometric programming: a programming approach to geometric design. *ACM Trans. Graph.*, 14(3):266–306, 1995.

[Pro10] Procedural Inc. Cityengine, www.procedural.com, 2010.

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

[Sha02] V. Shapiro. *Handbook of Computer Aided Geometric Design*, chapter 20: Solid modeling, pages 473–518. Elsevier, 2002.

[SHZO07] S. Sengupta, M. Harris, Y. Zhang, and J.D. Owens. Scan primitives for gpu computing. In *Graphics Hardware*, pages 97–106, NY, USA, 2007. ACM.

[SM78] G. Stiny and W. J. Mitchell. The palladian grammar. *Environment and Planning B*, 5:5–18, 1978.

[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.

[SS08] R. Schmidt and K. Singh. Sketch-based procedural surface modeling and compositing using Surface Trees. *Computer Graphics Forum*, 27(2):321–330, 2008. Proceedings of EG 2008.

[SS09] M. Schwarz and M. Stamminger. Fast gpu-based adaptive tessellation with cuda. *Computer Graphics Forum*, 28(2):365–374, 2009.

[STdKB10] R. Smelik, T. Tutenel, K.J. de Kraker, and R. Bidarra. Integrating procedural generation and manual editing of virtual worlds. In *PCGames '10: Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pages 1–8, New York, NY, USA, 2010. ACM.

[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. *Environment and Planning B: Planning and Design*, 9(1):113–114, 1982.

[VABW09] C.A. Vanegas, D.G. Aliaga, B.Beneš, and P.A. Waddell. Interactive design of urban spaces using geometrical and behavioral modeling. *ACM Trans. Graph.*, 28(5), 2009.

[VAW+10] C. Vanegas, D.G. Aliaga, P. Wonka, P. Müller, P. Waddell, and B. Watson. Modeling the appearance and behavior of urban spaces. *Comput. Graph. Forum*, 29(1):25–42, 2010.

[WMWG09] B. Weber, P. Müller, P. Wonka, and M.H. Gross. Interactive geometric simulation of 4d cities. *Comput. Graph. Forum*, 28(2):481–492, 2009.

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

[XFT+08] Jianxiong Xiao, Tian Fang, Ping Tan, Peng Zhao, Eyal Ofek, and Long Quan. Image-based façade modeling. *ACM Trans. Graph.*, 27(5):1–10, 2008.

[XFZ+09] Jianxiong Xiao, Tian Fang, Peng Zhao, Maxime Lhuillier, and Long Quan. Image-based street-side city modeling. In *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, pages 1–12, New York, NY, USA, 2009. ACM.

[YHL+07] T. Yang, Z. Huang, X. Lin, J. Chen, and J. Ni. A parallel algorithm for binary-tree-based string rewriting in l-system. In *Proc. of the Second International Multi-symposiums of Computer and Computational Sciences*, pages 245–252, Los Alamitos, California, 2007. IEEE Computer Society Press.

[YKG09] K. Yue, R. Krishnamurti, and F. Gobler. Computation-friendly shape grammars. In *Proceedings of CAAD futures*, pages 757–770, 2009.

[ZHW+06] K. Zhou, X. Huang, X. Wang, Y. Tong, M. Desbrun, B. Guo, and H.Y. Shum. Mesh quilting for geometric texture synthesis. *Proc. of ACM SIGGRAPH 2006*, 25(3):690–697, 2006.

[ZHWG08] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5):1–11, 2008.

Curriclulum vitae

Name: Markus Lipp

Date of birth: 15. 10. 1981

Nationality: Austrian

Email: lipp@cg.tuwien.ac.at

Languages: German, English

Education

Abbreviations: VUT = Vienna University of Technology

1997–2001: Technical High-School, main focus Telecommunications (HTL Nachrichtentechnik) in Innsbruck

June 2001: Graduation (Matura) (with distinction) from HTL Innsbruck

2001–2010: Studies in Computer Science at the VUT with special emphasis on Computer Graphics

April 2005: Graduation in Computer Science as “Bakkalaureus der Informatik” from the VUT

September 2007: Graduation in Computer Science (with distinguished honours) as “Diplom-Ingenieur der Informatik” from the VUT (thesis: “Interactive Computer Generated Architecture”)

Since October 2007: Phd student at the Institute of Computergraphics and Algorithms

Employment

Abbreviations: WS= winter semester; SS = summer semester;

Summer 1998-2000: Practica at Electronic Works Reutte (EWR) working as electrician, one month each summer

Summer 2001-2004: Heavy shift worker at Plansee metallurgical products, one month each summer

February 2003: Website developer at Steel & Metals Market research

Oct 2004–Apr 2005: Practica at the Institute for Computer Graphics on “High Dynamic Range Rendering” (VUT)

Sept 2005–Oct 2005:	Practica at VRVis research company on “Precalculated Global Illumination”
WS06:	Tutor in computer graphics 2+3(VUT)
SS07:	Tutor in real-time graphics (VUT)
Since October 2007:	Research assistant at the FIT-IT project “Gameworld”’ ’ (VUT)