HAL
open science

Interactive generation and rendering of massive models : a parallel procedural approach

Cyprien Buron

To cite this version:

Cyprien Buron. Interactive generation and rendering of massive models : a parallel procedural approach. Other [cs.OH]. Université de Bordeaux, 2014. English. NNT : 2014BORD0014 . tel-00988387v2

HAL Id: tel-00988387

https://theses.hal.science/tel-00988387v2

Submitted on 25 Aug 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

présentée à

L'UNIVERSITÉ de BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Cyprien Buron**

Pour obtenir le grade de

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Interactive Generation and Rendering of Massive Models: a Parallel Procedural Approach

---

Soutenue le : 04/02/2014

Devant la commission d'examen composée de :

Stéphane Mérillou	Professeur, Univ. Limoges	Président
Jean-Michel Dischler	Professeur, Univ. Strasbourg	Rapporteur
Eric Galin	Professeur, Univ. Lyon 2	Rapporteur
Gael Guennebaud	CR Inria, Univ. Bordeaux 1	Examinateur
Xavier Granier	Professeur, Univ. Bordeaux 1	Directeur de thèse
Jean-Eudes Marvie	PhD, Principal Scientist, Technicolor	Encadrant de thèse

– 2014 –

Remerciements

Première page à lire, mais dernière page rédigée. C'est enfin l'occassion de remercier toutes les personnes qui m'ont aidé et accompagné durant ces 3 passionnantes années de thèse.

Tout d'abord, je souhaite adresser mes premiers remerciements à mes directeurs de thèse : Jean-Eudes Marvie et Xavier Granier. Merci de m'avoir m'avoir fait confiance et donné l'opportunité de réaliser cette thèse en collaboration entre Technicolor et Manao. Je vous suis très reconnaissant de m'avoir encadré, orienté, et conseillé durant ces années.

J'adresse mes remerciements à Jean-Michel Dischler et Eric Galin pour avoir lu avec attention mon manuscript et à Stéphane Mérillou pour avoir présidé mon jury de soutenance.

Je tiens également à remercier l'équipe Manao de m'avoir accueilli lors de mes séjours à Bordeaux, et Xavier pour m'avoir héberger. Plus particulièrement, j'adresse un grand merci à Gaël Guennebaud pour toute l'aide apportée et son implication dans mes travaux de recherche.

Je souhaite également remercier l'équipe à Technicolor notamment Gaël, Pascal, Patrice, Olivier, Alex, sans oublier Pascal Gautron et Sandra. La bonne ambiance toujours présente dans l'équipe a permis un cadre de travail idéal. Les aides, remarques et discussions fructueuses avec chacun ont grandement contribué au bon déroulement et à la réussite de cette thèse. Notamment, les devs avec Pat, les démos et les rushs vidéos de P'ti Fourbe, les relectures toujours constructives de Jean-Eudes, Xavier, des Pascals et des Gaëls (et leurs stylos rouges tombés au combat) ont permis ces travaux. Les soirées deadline où bonne humeur régnait avec travaux de dernières minutes ont toujours été de grands moments.

Enfin, j'adresse un grand merci à mes parents et à ma femme Stéphanie pour m'avoir encouragé, soutenu et supporté depuis le début.

Contents

1 Introduction **1**

2 Parallel computing & graphics hardware **5**
2.1 Theory of parallel computing 5
2.1.1 Power consumption issue 6
2.1.2 Theoretical speedup 6
2.1.3 Parallelism conditions 9
2.1.4 Parallel system classification 9
2.1.5 Parallel programming types 11
2.1.6 Parallel hardware 11
2.2 Graphics processing unit 12
2.2.1 History 12
2.2.2 Unified pipeline revolution 13
2.2.3 Hardware tessellator 16
2.3 Logical graphics pipeline - DirectX 11 / OpenGL 4 19
2.4 Conclusion 22

3 Procedural content generation **23**
3.1 Introduction to procedural generation of contents 23
3.2 Grammar-based modeling of plants 26
3.3 Grammar-based modeling of buildings 29
3.4 Geometry synthesis on surfaces 32
3.5 Improving artist experience 34
3.6 Real-Time procedural generation on GPU 37
3.7 Conclusion 39

4 Parallel procedural generation based on independent 1D atoms **41**
4.1 Introduction 42
4.2 Procedural pipeline overview 43

4.3 Rule compiler 44
4.4 Rule expander 46
4.4.1 Grammar-specific GPU expression-instantiated interpreter 46
4.4.2 Segment-based expansion 47
4.4.3 Implementation on graphics hardware 51
4.5 Terminal evaluator 53
4.5.1 Renderer 54
4.6 Applications 54
4.6.1 Architecture 55
4.6.2 Vegetation 55
4.6.3 Object batching & performance 58
4.7 Conclusion 61

5 Internal context parallelization
and application to roofs modeling **63**
5.1 Introduction 64
5.2 Principle 64
5.2.1 Internal consistent context computation 66
5.2.2 Internal context decomposition 66
5.2.3 Join rule 67
5.2.4 Clipping 68
5.2.5 Project rule 69
5.3 Case study 70
5.4 Applications & results 72
5.5 Discussions & future works 73
5.6 Conclusion 75

6 External context-sensitive grammars
and application to growth on generic shapes **77**
6.1 Introduction 78
6.2 Passing context through texture maps 79
6.3 Indirection geometry images 81
6.3.1 Geometry images 81
6.3.2 Indirection computation 82
6.4 Marching rule 84
6.4.1 Best matching pixel 87
6.4.2 Tangent-based orientation of the marching direction 88
6.5 Smooth seamless geometry synthesis 90
6.6 Applications & results 93
6.6.1 Ivy growth 94
6.6.2 Grammar constraints through texture contexts 96
6.7 Discussion 98
6.8 Conclusion 98

7 Parallel grammars scalability
and application to massive sceneries **101**
7.1 Introduction . . . 102
7.2 LOD system overview . . . 103
7.3 Architecture . . . 104
7.3.1 Impostors . . . 105
7.3.2 LODs description . . . 106
7.3.3 LOD transition . . . 108
7.4 Vegetation . . . 108
7.4.1 Impostors . . . 108
7.4.2 LOD description . . . 110
7.5 Caching, clustering and culling . . . 110
7.5.1 Caching . . . 110
7.5.2 Clustering . . . 110
7.5.3 Fine-grain culling on GPU . . . 111
7.6 Results . . . 112
7.7 Conclusion . . . 115

8 Conclusion **117**
8.1 Contributions . . . 117
8.1.1 Parallel procedural generation based on independent 1D atoms . 117
8.1.2 Internal context parallelization . . . 118
8.1.3 External context-sensitive grammars . . . 118
8.1.4 Parallel grammars scalability . . . 118
8.2 Perspectives . . . 118

9 Résumé en Français **121**
9.1 Introduction . . . 121
9.1.1 Contributions . . . 122
9.2 État de l'art sur la génération procédurale . . . 122
9.3 Génération procédurale en parallèle basée sur des atomes 1D indépendants 123
9.3.1 Description du pipeline . . . 124
9.3.2 Applications et résultats . . . 125
9.4 Parallélisation du contexte interne . . . 127
9.4.1 Méthode . . . 127
9.4.2 Application à la génération de toits & résultats . . . 128
9.5 Grammaires sensibles au contexte externe . . . 129
9.5.1 Méthode . . . 129
9.5.2 Application à la croissance sur des surfaces génériques & résultats 130
9.6 Mise à l'échelle des grammaires parallèles . . . 131
9.6.1 Méthode . . . 131
9.6.2 Application aux environnements massifs & résultats . . . 132
9.7 Conclusion . . . 132
9.7.1 Contributions . . . 132

9.7.2 Perspectives 133

A Grammar rule library **137**

Bibliography **143**

Chapter 1

Introduction

Movie and video game industries endlessly create more huge and more complex environments in order to render even more realistic scenes. However, the natural elements and architectural models composing the sceneries, such as trees and buildings, are orders of magnitude too complex to be modeled and stored for large-scale environments. In addition, the generated models being extremely huge and complex, rendering an entire scene or editing one or many objects within the environment becomes a very time-consuming and tedious task. These drawbacks prevent from interactive pre-visualization and fast production.

In this thesis, we aim at generating and rendering the elements composing massive scenes at interactive time. To do so, we propose a parallel procedural approach, benefiting from the highly parallel computation capabilities of recent graphics hardware.

Procedural methods provide tools for quickly modeling elements exhibiting repetitive patterns like buildings and trees. Especially, grammar-based methods use a set of rules and parameters in order to describe the lightweight structure of the objects. Starting from a small set of rules, an infinite number of unique but similar objects are generated depending on the input parameters. Grammar-based methods, through their amplification role, open new perspectives in modeling large-scale environments. For instance, the New York cityscape of the *King Kong* movie is generated using *CityBot* [Whi06], the procedural modeling system developed by *Weta Digital*. The large-scale buildings of *Man of Steel* are created with *CityEngine* [Esr12].

Traditionally, procedural approaches generate the amplified models using the Central Processing Unit (CPU), and transfer the results to the memory of the Graphics Processing Unit (GPU) for later rendering. Pre-amplification leads to memory consumption and geometric complexity issues, preventing from real-time edition for large-scale models. We thus explore the parallelism theory to generate and render massive scenes interactively.

With the increasing parallel computation power of recent graphics hardware, parallel

computing on GPUs became a trend research topic. For instance, global illumination, physics simulation and image processing successfully took advantage of the GPU. The high number of computing cores in such architectures allows for fast computation on very large datasets.

Porting sequential programs to parallel architectures requires to rethink the algorithm, and deal with the parallelism constraints. Our contributions bring the parallelism paradigm to the procedural approach in order to reach interactive performances for huge procedural environments.

Contributions

We take advantage of both parallelism and procedural generation of highly detailed models by introducing a new parallel procedural approach. We extend our approach to deal with incoherent generated models, and massive environments. Four main contributions are proposed in this thesis:

- We design GPU shape grammars: a parallel procedural pipeline for interactive generation and rendering of highly detailed objects
- This pipeline is extended to consistently generate models relying on internal contexts, such as roofs
- External context sensitive grammars are introduced for consistent expansion on surfaces and grammar constraints through texture maps
- Finally, we propose parallel grammars scalability to address massive environments

Our parallel procedural approach is based on the decomposition of input objects into independent 1D atoms. Thanks to our segment-based expansion, each 1D atom is able to evaluate the grammar in a first parallel stage. Then, geometric terminal shapes are instantiated in a second parallel stage. Finally, the rendering of the generated highly detailed models is performed consistently to existing rendering pipelines. We demonstrate the efficiency of our pipeline with interactive modeling of both vegetation and architecture.

Decomposition of the input objects into 1D atoms results in independent expansion of each atomic element. To model internally consistent objects, we extend our pipeline to take care of internal contexts with respect to parallelism. Our method is illustrated on interactive consistent roof generation over multiple independent atoms.

Grammar rules are often controlled by non-intuitive parameters. In order to constrain the grammar rules with more artist-friendly tools, our pipeline integrates external context-sensitive grammars. Texture contexts such as pre-generated texture maps or on-the-fly paintings are simple way to control the grammar behavior. We also introduce surface context for consistent expansion onto an underlying surface. External context-sensitive grammars are particularly useful to interactively generate growth on generic shapes.

While GPU shape grammars pipeline handles hundreds of grammar-based models interactively in parallel, we bring scalability to parallel grammars in order to scale up to thousands of generated models at interactive times. To do so, we introduce a complete LOD system based on advanced rendering optimization techniques within our pipeline. Massive sceneries are then supported with interactive feedback for artists.

Thesis overview

This thesis is divided into 7 chapters. We first introduce essential concepts about the parallelism paradigm, we detail the specific parallelism of the graphics hardware and review the graphics pipeline in Chapter 2. Related works on procedural generation are presented in Chapter 3. The next four chapters present our contributions. From Chapter 4 to Chapter 7 we introduce respectively the parallel procedural approach used, the internal context parallelization, external context sensitive grammars and the parallel grammars scalability. Finally, Chapter 8 concludes this thesis and opens some research perspectives.

Chapter 2

Parallel computing & graphics hardware

This chapter introduces the theory of parallel computing. We start with a quick overview of parallelism, essential for a good comprehension of the research work of this thesis. For more details about parallel computing, we suggest you to refer to [KWm10]. Then we focus on the parallelism capabilities of recent graphics processing units (GPU). Finally we address the logical graphics pipeline according to the DirectX® 11[1] / OpenGL® 4[2] specifications.

2.1 Theory of parallel computing

Traditionally, computer softwares are designed for sequential computing. In order to resolve a problem, one may create an algorithm, and implement it with a stream of serial instructions. Only one instruction at a time is executed on a CPU (Central Processing Unit). Once execution is finished, the next instruction is processed and so on. On the other hand, in parallel computing, a problem is broken into independent tasks. Each one may be solved concurrently by an algorithm composed of sequential instructions. Dividing a problem into independent smaller ones takes advantage from multiple processing units. Each processing unit available can execute its part of the algorithm simultaneously. Parallelism has been used for around 30 years in high-performance computing but supporting hardware were very expensive, thus limiting its popularity. However, since the apparition of many-cores GPUs, usable for general parallel computing inside everyday usage computers, any application can benefit from parallel programming.

[1] http://msdn.microsoft.com/en-us/library/windows/desktop/ff476080(v=vs.85).aspx
[2] http://www.opengl.org/registry/

2.1.1 Power consumption issue

Runtime of computer software mainly relies on the speed of the processing unit: the faster the processor executes an instruction, the faster is the application. Computation performance used to increase only with frequency scaling. However, power consumption depends on frequency. The equation of the chip consumption is given by:

$$P = C \times V^2 \times F \tag{2.1}$$

where P is the power consumption, C the capacitance of all transistors whose inputs change per clock cycle, V is the voltage of the chip, and F is the processor frequency in cycles per second. Increasing the frequency of the chip involves directly a higher power consumption and a heat dissipation issue. This is the main reason that the race frequency has been stopped at the beginning of 2004. However, Moore's law stating that the number of transistors in a chip doubles every 18 to 24 months [M+65] is still effective because additional transistors are now used for duplicating processors for parallel computing.

2.1.2 Theoretical speedup

Amdahl's law is a model showing the speedup of a parallelized algorithm according to a given number of threads running in parallel and a given portion of algorithm that cannot be parallelized [Amd67]. Assuming the algorithm is a problem of a fixed size no matter the parallelization, we can compute the maximum speedup independently from the number of threads assigned to the computation. Let $n \in \mathbb{N}$ be the number of threads for computation, $B \in [0; 1]$ the fraction of the algorithm remaining serial, the theoretical speedup of computation is:

$$S(n) = \frac{1}{B + \frac{1-B}{n}} \tag{2.2}$$

The maximum speedup reachable when n tends to infinity is:

$$\lim_{n \to +\infty} S(n) = \frac{1}{B} \tag{2.3}$$

The theoretical maximum speedup only depends on the portion of the algorithm that cannot be parallelized, and is independent from the number of threads assigned as shown in Figure 2.1.

Another interpretation from Equation 2.3 is that in a sequential algorithm divided into many parallel tasks, it is more efficient to achieve a small parallelism onto the largest fractional part than applying big parallelism onto the smallest ones. For instance, let us consider a program composed of two independent tasks A and B, A taking 75% of the computation time, and B 25% (Figure 2.2). If we achieve a speedup of $\times 5$ for the computation of the task B, we will reach a total speedup of 1.25. On the other hand, just achieving a speedup of $\times 2$ for the task A allows for a better total speedup of 1.60.

While Amdalh's law assumes a problem of fixed size, Gustafson's Law considers problems of large scale where the data sets may grow with the number of processors

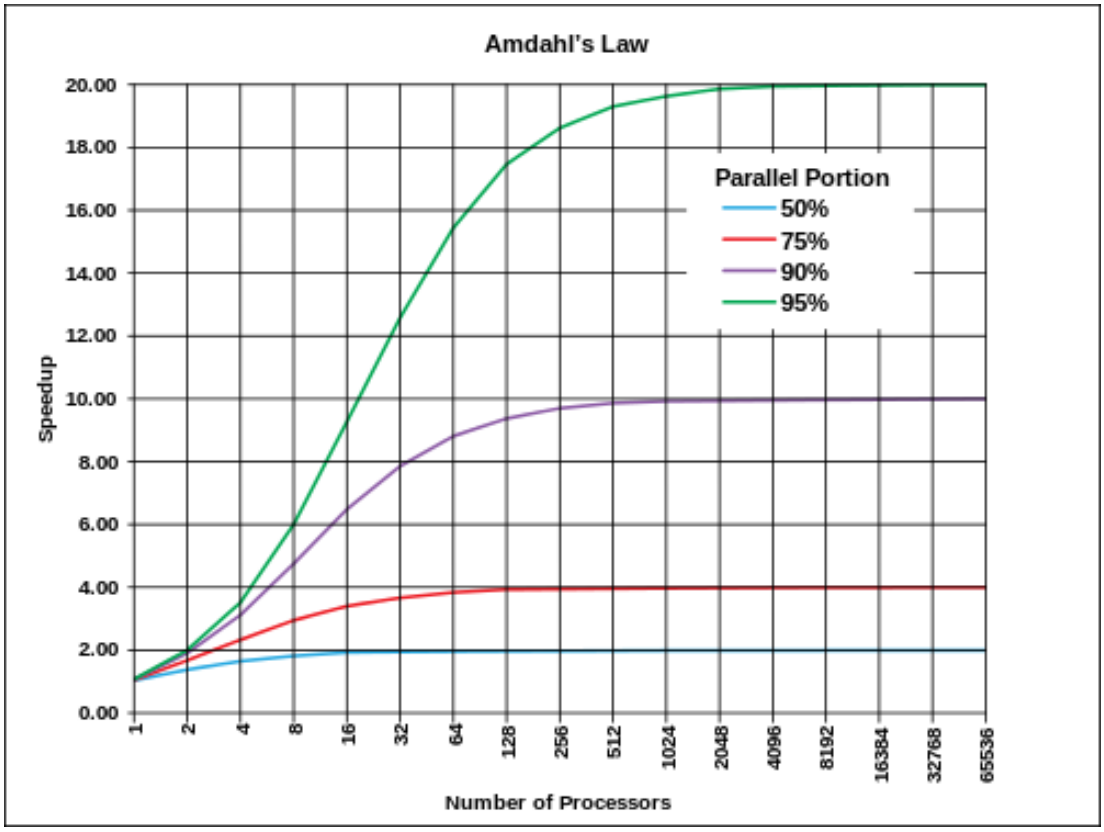
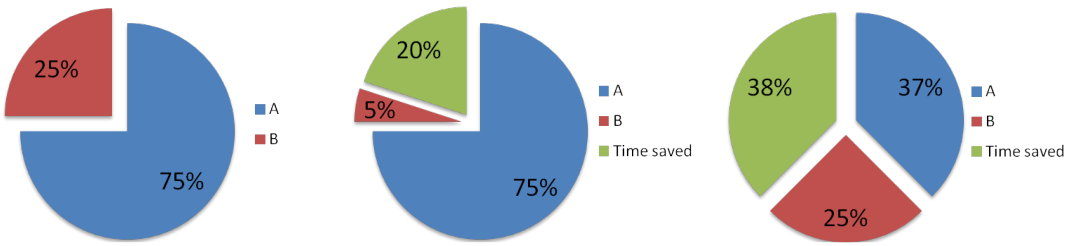


Figure 2.1: Theoretical maximum speedup does not depend on the number of threads working in parallel, only on the fractional part of the algorithm parallelizable. Image courtesy of Wikipedia.



(a) Original time repartition (b) B 5× faster (c) A 2× faster

Figure 2.2: Focusing efforts on paralleling larger parts of an algorithm is more profitable than working hard to achieve a better speedup onto the smaller parts. Here a speedup of ×2 for the computation of the part A gives a better total speed computation improvement (c) than a speedup of ×5 for the part B (b).

[Gus88]. Contrary to Amdalh's law stating that the maximum speedup reaches a limit with a large number of threads working in parallel, Gustafon's law says that the speedup always grows linearly with the number of threads and the input data to work on. However, accordingly to Amdalh's law, the smaller the serial parts of the algorithm, the faster the computation. Considering a problem with a sequential fractional part A, the speedup is described as:

$$S(P) = P - A \times (P - 1) \tag{2.4}$$

where P is the number of processors assigned to the computation.

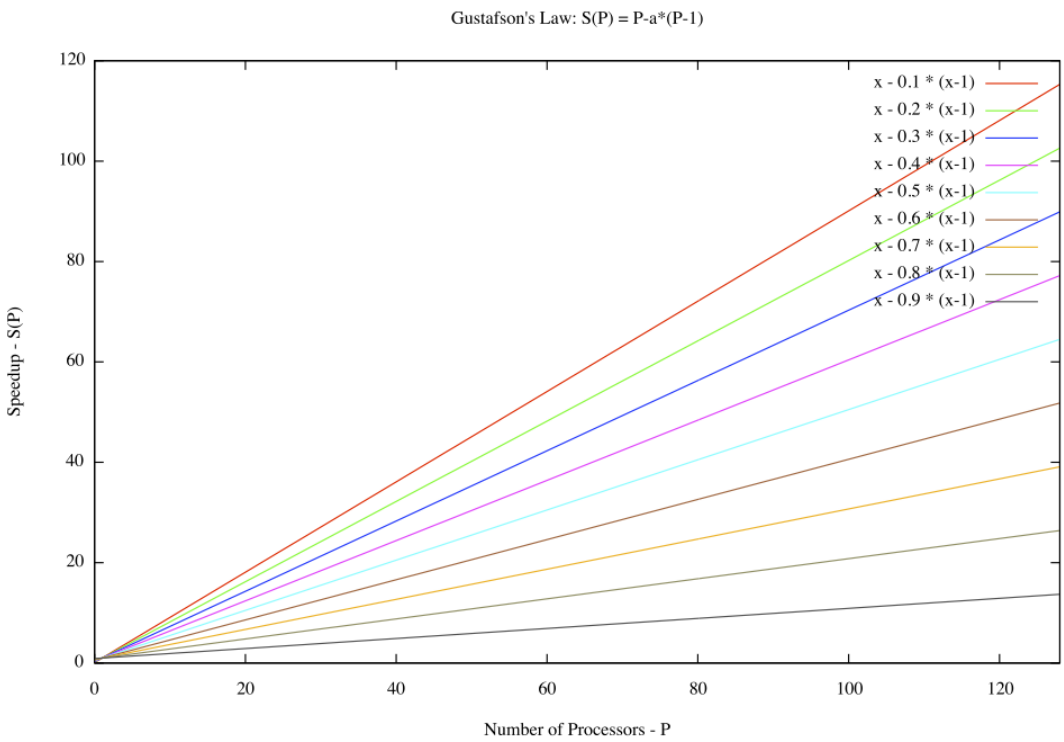


Figure 2.3: For large input data, maximum speedup of parallelization grows linearly according to the number of processors. Image courtesy of Wikipedia.

Figure 2.3 shows different speedup cases. The equation 2.4 says that the computation time of the serial part of the algorithm does not change no matter the number of processors. If it takes 1s to compute the sequential part with 1 processor, it will always take 1s even with 100 processors. Conversely, the parallel part will have a speedup proportional to the number of processors because we can compute the parallel part of many input data in the same amount of time than for only one input.

Both laws address different problems, Amdalh's law for problem of fixed size while Gustafson's law generalizes to arbitrary data sets. By associating both laws, me may benefit from both the parallelization of non sequential parts of a program, and the

parallelization over the data sets. Our contributions are designed respectfully to this observation.

2.1.3 Parallelism conditions

An algorithm composed of multiple program segments can be computed in parallel if Berstein's conditions are satisfied [Ber66]. Let P_i be the program segment i, with input and output variables I_i and O_i respectively. Likewise for the program segment P_j. P_i and P_j are independent and can be computed in parallel if:

$$I_j \cap O_i = \varnothing, I_i \cap O_j = \varnothing, O_i \cap O_j = \varnothing \tag{2.5}$$

The two first conditions state the input of one program segment must not depend on the output of the other program segment. The third condition says the two program segments must not write outputs at the same location. If one of the conditions failed, it shows a flow dependency and program segments are not independent. If successful, they can be processed in parallel. However while Berstein's conditions prohibit shared memory between program segments, it remains possible to achieve parallelism using synchronization mechanism such as semaphores, mutual exclusions, barriers, etc.

2.1.4 Parallel system classification

Processor architectures may be classified into different categories. One of the most widely used is the Flynn's taxonomy [Fly66]. It differentiates multi-processor architectures according to the number of concurrent instruction data streams. Four categories are defined as shown in Figure 2.1:

- SISD: Single Instruction, Single Data. Only one instruction is executed by the CPU at any clock time. Only one data is used at any clock time.
- SIMD: Single Instruction, Multiple Data. All processing units execute the same instruction at any clock time, but on different data. For instance a vector addition can be performed in only one SIMD instruction where the multiple data are the four components of the vector.
- MISD: Multiple Instruction, Single Data. All processing units execute different instructions at any clock time, on the same data.
- MIMD: Multiple Instruction, Multiple Data. All processing units execute different instructions at any clock time, on different data.

On top of this classification of processor architectures, two sub-categories of the MIMD are defined: SPMD and MPMD. In Single Program Multiple Data mode, multi-processors executes the same program but independently. As these multi-processors do not share the same program counter, executions are independent. Multiple Program Multiple Data is the extension of the SPMD paradigm with various programs in input.

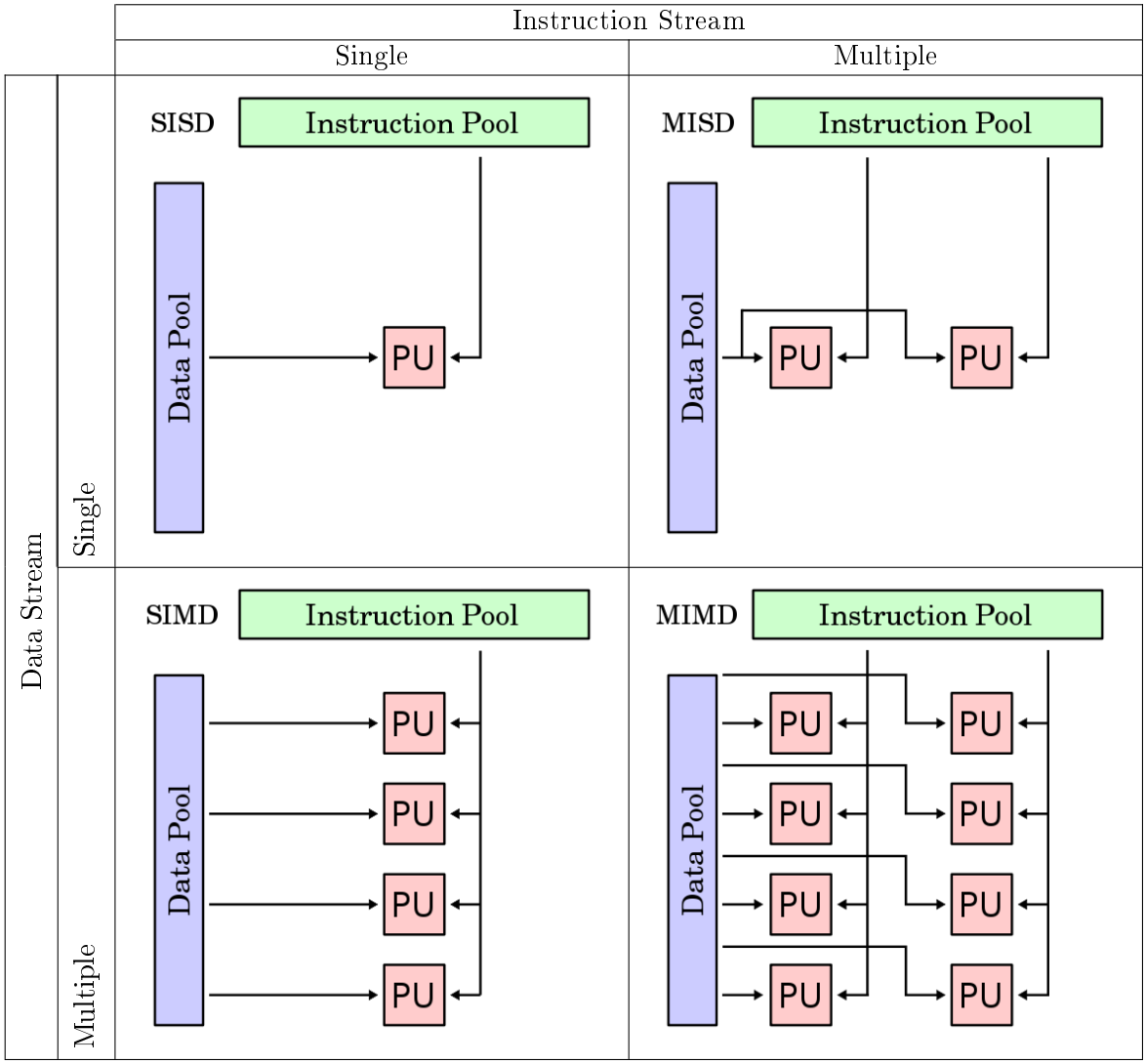


Table 2.1: Flynn's taxonomy differencing computer according to instruction and data streams, processed by Processing Units (PUs). Courtesy of Wikipedia.

2.1.5 Parallel programming types

Different programming types exist for parallel computing: bit-level parallelism, instruction-level parallelism, task parallelism and data parallelism. We will not go into details for the first three categories, but only for the one that interests us here: the data parallelism.

Data parallelism focuses on a collaborative distributed work on a same data set. Each thread will have a specific partition of the data set assigned and threads perform the same task on their own data partition. For instance, a matrix addition can be parallelized through data parallelism. Having two 1000×1000 matrix in input, A and B, the addition of A plus B is computed into an output 1000×1000 matrix C. Depending on the number of available threads N, the work may be subdivided into N arbitrary sub-partitions (Figure 2.4). Instead of having only one thread performing the 1000000 additions, we take advantage of the N threads, each computing $\frac{1000 \times 1000}{N}$ additions. Ideally same result is achieved N times faster using data parallelism.

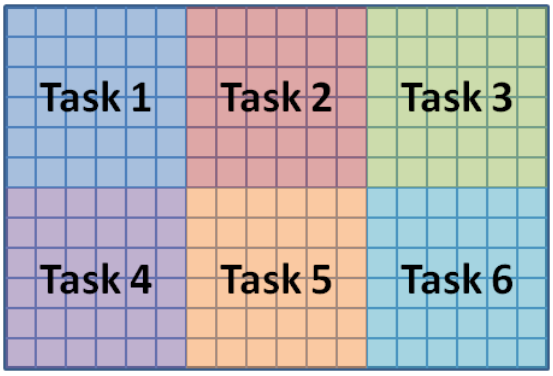


Figure 2.4: A 12×18 matrix is partitioned into 6×6 subsets. Each thread is assigned a subset of the work and is executed in parallel.

We focus onto data parallelism because we extensively use data parallel computation type for this research work. Moreover as we will see in the next subsection, GPU is designed especially for this kind of parallelism and is extremely powerful at executing such computation in parallel. Furthermore, GPU processing units also support SIMD computations. For instance, addition of two vectors may be realized with only one vector addition instruction processed on each component of the vectors.

2.1.6 Parallel hardware

Different classes of hardware exist for parallel computing: multi-core computing, distributed computing, cluster computing, massive parallel processing, grid computing, GPUs, and some others. We focus on only two parallel classes of hardware, the multi-core computing and the GPU because these hardwares are now quite cheap and very widespread inside industries and prosumers, contrary to highly expensive hardware such as massive parallel processing or grid computing.

Multi-core processors refer to CPUs available since 2004/2005 for general audience

where a processor is composed of up to around ten cores integrated into the same chip. Each core has its own instruction stream, and may execute independent operations: multi-core processors belong to the MIMD class. However, with the SSE instructions available on many CPUs, they are able to perform SIMD instructions at a low-level. On the other hand, GPUs are now used for general-purpose computing. Contrary to the multi-core processors, GPUs are stream processors that allow each core to execute a same instruction onto a different data set (SIMD). For instance, vectorial computations are performed in parallel for each element of the vector. In addition, recent GPUs are also categorized as SPMD because each core can follow different paths through the same code. They are heavily optimized for data parallelism, and generally contain hundreds of cores. We may compare two current high-performance CPU and GPU in order to see the divergence of number of computing cores. For instance, the CPU *Intel® Xeon® E7-8870* has 10 cores composed of 8 computing units each, giving 80 total computing units whereas the GPU *NVIDIA® GeForce® GTX 780* has 2304 computing cores.

However, the number of cores does not make a GPU always more efficient than a CPU. Graphics hardware has also some limitations, for instance it has less cache memory than a CPU, and is not as comfortable with complex instructions (branching, loop, etc) as the CPU. Globally we can say that the GPU is better for simple calculation on many data, whereas the CPU is better for complex computation on few data.

2.2 Graphics processing unit

Graphics Processing Units are designed to reduce the CPU load by performing the operations related to the computation and display of synthetic images, that we call graphics rendering. Recently GPUs have also been used in more high-level processing: the general-purpose computing on graphics processing units (GPGPU). Both approaches take advantage of the highly parallel structure of the GPU. In this section we describe into more details the architecture of a recent GPU, which is essential to understand the work of this thesis. In order to see how a same hardware can be used for both graphics rendering and general-purpose computing, we start by a brief review of graphics processing unit history. Then we see its evolution to the unified pipeline. Finally we describe an interesting feature of recent GPUs: the hardware tessellator.

2.2.1 History

Historically first GPUs have been developed for graphics rendering purpose. Some logical graphics pipeline have been designed (DirectX 7, DirectX 8, etc) directly affecting hardware architecture. Let's take the example of the DirectX 7 specifications. Figure 2.5 shows the classic pipeline used for many years. Starting from the top, where data is fed from the CPU to the GPU, multiple processing stages are executed to finally get a pixel drawn on the screen. Basically the GPU receives vertex data (position, normal, texture coordinates and color) from the CPU and these data are then processed by the

vertex stage. This stage performs a fixed-function[3] transform and lighting operation per vertex. The next step is the primitive assembly stage where vertices are assembled into primitives such as triangles, lines, or points. These primitives go through the rasterization stage. It converts primitives into pixel fragments. Then the fixed-function fragment step computes shading for each fragment. Finally the raster operations (ROP) eliminates invisible fragments according to Z-testing, it also computes blending between fragments if needed and anti-aliasing. The visible fragments are then written into the frame buffer. Over the years, the classic pipeline remained mainly unchanged. Most notable evolutions were vertex shader programmability brought with DirectX 8, and similarly for fragment shader with DirectX 9. However this graphics pipeline was nearly hardware-implemented as-it-is. Each stage of the pipeline was processed by a specialized physical part of the graphics hardware, following the linear pipeline scheme. In fact, each major processing stage of the pipeline was composed of many physical sequential stages. For example, in *GeForce 7* GPUs fragment shader stage usually had more than 200 sequential hardware stages. Implementation of this classic graphics pipeline resulted into a high number of physical stages sequentially ordereds.

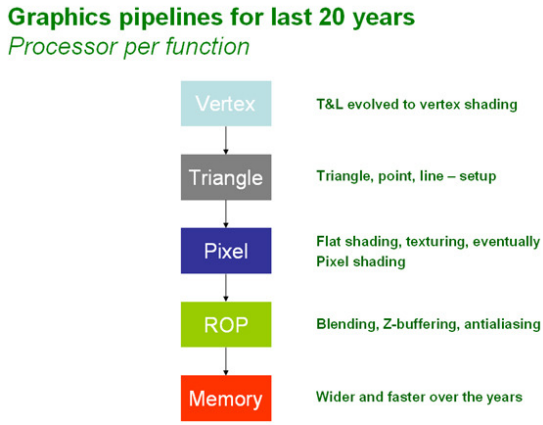


Figure 2.5: Classic graphics pipeline before unified pipeline architecture. Courtesy of NVIDIA.

2.2.2 Unified pipeline revolution

With *G80* architecture, NVIDIA decided to change their philosophy of the hardware implementation of the graphics pipeline. While previous approach was a sequential hardware pipeline directly simulating graphics pipeline, they designed a unified pipeline and shader architecture where the sequential pipeline flow is modified to be more looping oriented (Figure 2.6). The new unified shader processors are able to perform any shader computation (vertex, fragment), and it can feed back itself result of current operation to execute another one. This new approach significantly reduces the number of

[3] A fixed-function operation is a stage hardware implemented that may be parametrized but not modified.

hardware pipeline stages. Such an architecture was first hardware-implemented within the *NVIDIA GeForce 8800* graphics card (Figure 2.7).

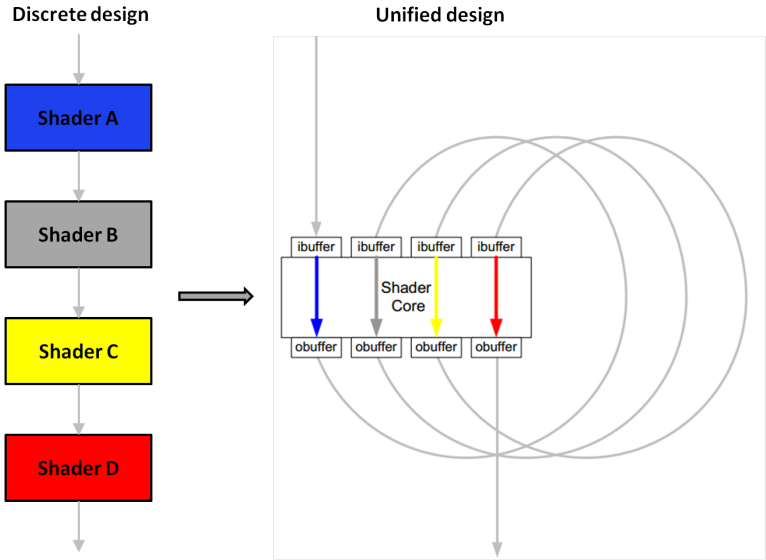


Figure 2.6: In unified design (since G80 architecture), a single core can execute each shader stage. Courtesy of NVIDIA.

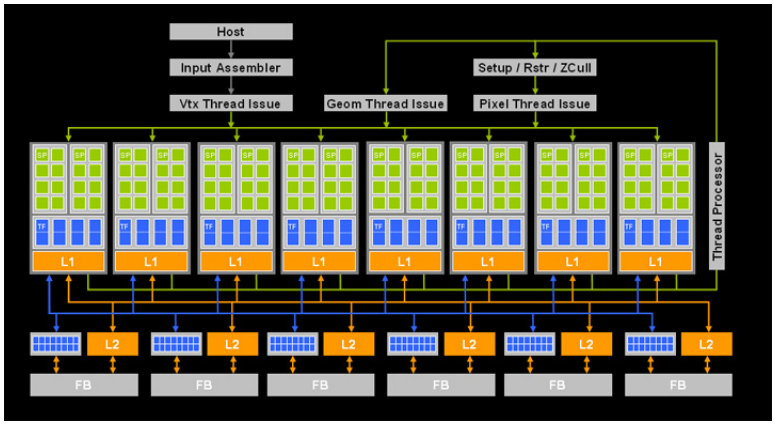
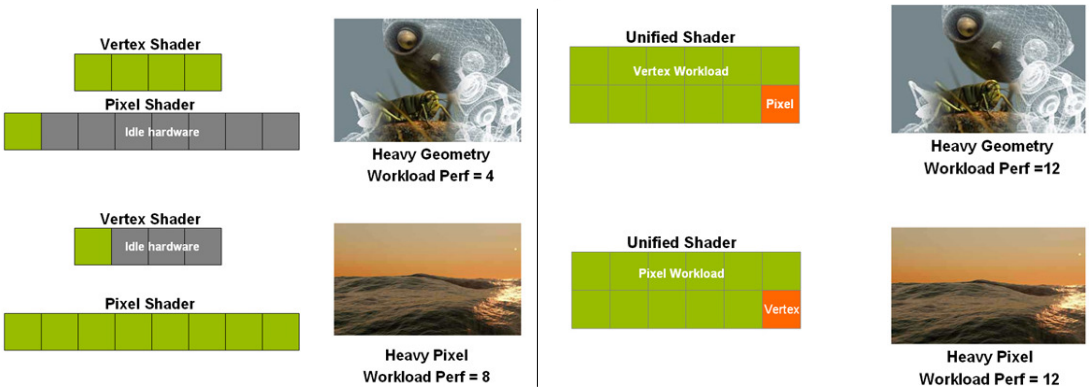


Figure 2.7: G80 architecture based on Streaming Multiprocessors. Courtesy of NVIDIA.

This radical modification of architecture has been motivated by the following observation. Some graphics applications may be vertex shader intensive, while others may be pixel shader intensive, especially for recent programs. Workload for vertex and pixel shaders may be completely imbalance at any given time of the computation. For instance let us imagine a GPU with a fixed number of shader units: 4 vertex shader cores, and 8 pixel shader cores (Figure 2.8a). In case of a vertex shader intensive application, the maximum performance is reach when all vertex shader units are used, so it corre-

sponds to 4 in our example. On the other hand for fragment shader intensive scene, maximum performance is limited to number of fragment shader units available, here 8. In both case, a bottleneck appears because shader work is not well distributed among shader units. In addition, we also observe that some shader units are not working at all and are idle. The idea of unified pipeline and shader architecture came from this observation: how can we fully take advantage of all shader units available at any time? Unified shader processors should be generic so that they could perform any shader invocation at any time, in order to minimize idle time. This is illustrated in Figure 2.8b. In vertex shader intensive case, most of the shader units are used for vertex computation, attaining a performance score of 12, and accordingly for fragment shader intensive case. In both cases, all shader cores are used, maximizing computation performance. This fundamental pipeline modification brought general-purpose computing to graphics hardware, while also enhancing graphics pipeline with a better load balancing. The same hardware allows us to do either parallel general-purpose computing (physics simulation, image processing, etc) or parallel graphics rendering (image synthesis, etc) using the same computation cores.



(a) Specialized shader stages (b) Unified shaders

Figure 2.8: Unified computing shader units allow to achieve better performances in any application case. Courtesy of NVIDIA.

This unified pipeline has been the starting point of all recent graphics pipeline architecture such as *NVIDIA Fermi* and *NVIDIA Kepler*. For instance, if we look at the *NVIDIA GeForce 680 GTX* specifications, corresponding to the *GK104 Kepler* architecture (Figure 2.9), we note that the computing cores (also called CUDA® cores) are regrouped into 8 streaming multiprocessor (SMX). Each SMX contains 192 computations cores, reaching a total number of 1536 computation cores. Work distribution is realized in a first time using a global scheduler distributing thread blocks to SMX, which then distribute them to their computation cores thanks to a warp scheduler.

For graphics rendering the scheduling is completely hidden from the programmer, but assure him an efficient load balancing. Cache management is also automatically

handled. Inversely, for general-purpose computing, programmer can choose the number of threads to launch, the thread block size, and how to control cache between local and global memory in order to get the maximum performance. At this step, we can say general-purpose computing allows a finer grained parallelization and a better cache management than for graphics rendering. However, when the final goal is to render an image, one should know that switching context from general-purpose computing to graphics rendering (for example when using OpenGL to render the final image depending on the results of an OpenCL® computation) is quite costly.

Most known languages for general-purpose computing on graphics hardware are CUDA and OpenCL. Programs that are processed on computation cores are called kernels. Similarly DirectX and OpenGL are used for graphics rendering, and programs executed are called *shaders*.

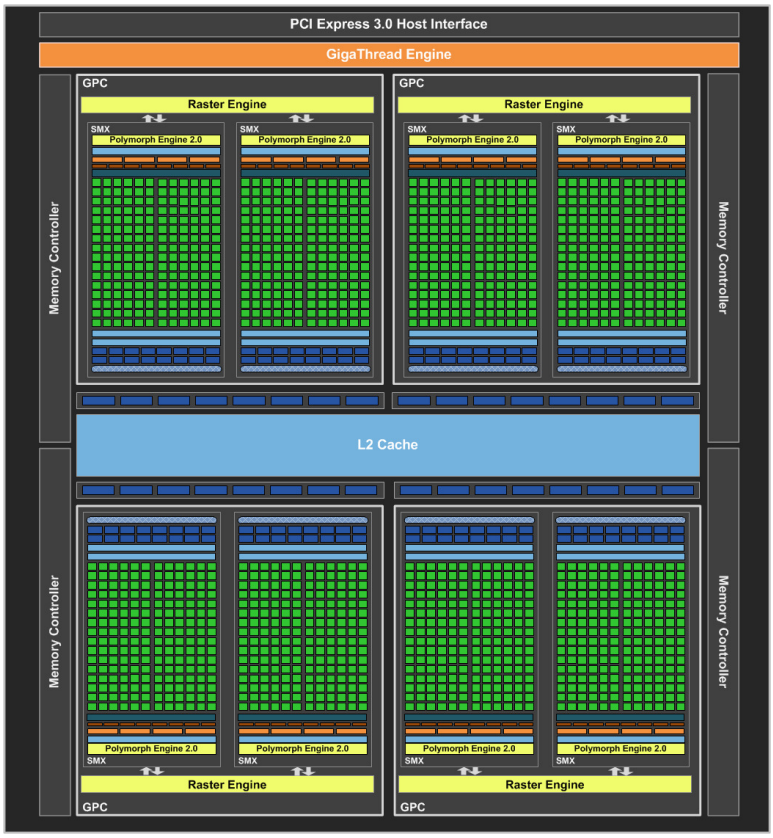


Figure 2.9: GeForce 680 GTX - GK104 Kepler Architecture.

2.2.3 Hardware tessellator

Kepler architecture and equivalent also added a special hardware unit responsible for tessellation, introduced in DirectX 11, called hardware tessellator. Tessellation is a process allowing creation of new vertices on top of existing triangles. It can amplify a low

polygon model into a denser one, while being smoother and smoother. Higher density models give higher quality rendering images, as the head model in Figure 2.10 shows. Before introduction of the hardware tessellation, in order to get highly detailed meshes, we had to tessellate the models using the CPU and then transfer the resulting objects to the GPU for rendering, which was very expensive operation. Then some GPU-based methods have been developed to refine meshes [BS05, SJP05, PO08, GBP11]. We may send only a low-resolution mesh to the GPU which will refine it on-the-fly and render it. This automatically accelerate mesh-based computations such as animations that can be done only onto the low-resolution mesh, before triangle amplification. Finally hardware tessellator increased refinement performance thanks to special hardware computing units dedicated for this operation avoiding multi-pass computation.

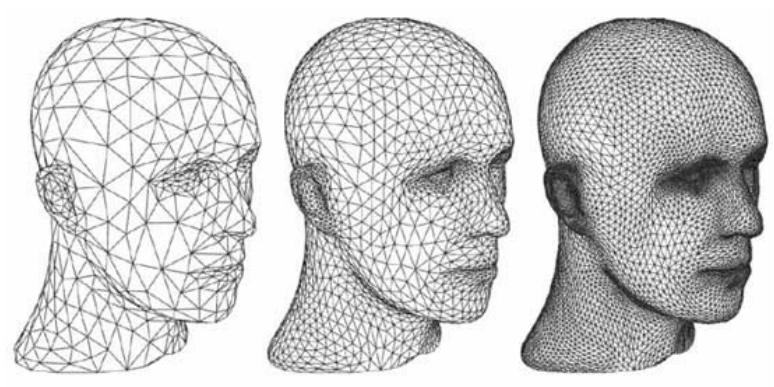


Figure 2.10: Tessellation on a head model.

Figure 2.11 shows the diagram of a streaming multiprocessor (SMX). We notice that hardware tessellator is a unit separated from computation cores. And as there is one tessellation core per SMX, we have as many tessellation units as the number of SMX. These new tessellation cores are specialized for quickly creating new vertices on-the-fly and directly stream them to the graphics pipeline. An important thing to note is that we may address these cores only using graphics rendering pipeline, not with general-purpose computing languages such as Cuda and OpenCL.

More recently OpenGL 4.2 introduced compute shaders. While being created for general-purpose computation, they slightly differ from CUDA/OpenCL kernels because they are directly integrated within logical graphics pipeline. They combine advantages of general-purpose programming with fine-grained parallelization and precise cache management but integration to graphics pipeline eliminates costly context switching. However they are only processed by computation cores and cannot address tessellation units.

As a major part of this thesis is about adding more and more details to models procedurally, we choose to implement all this work using graphics pipeline only in order to get intensive usage of hardware tessellator.

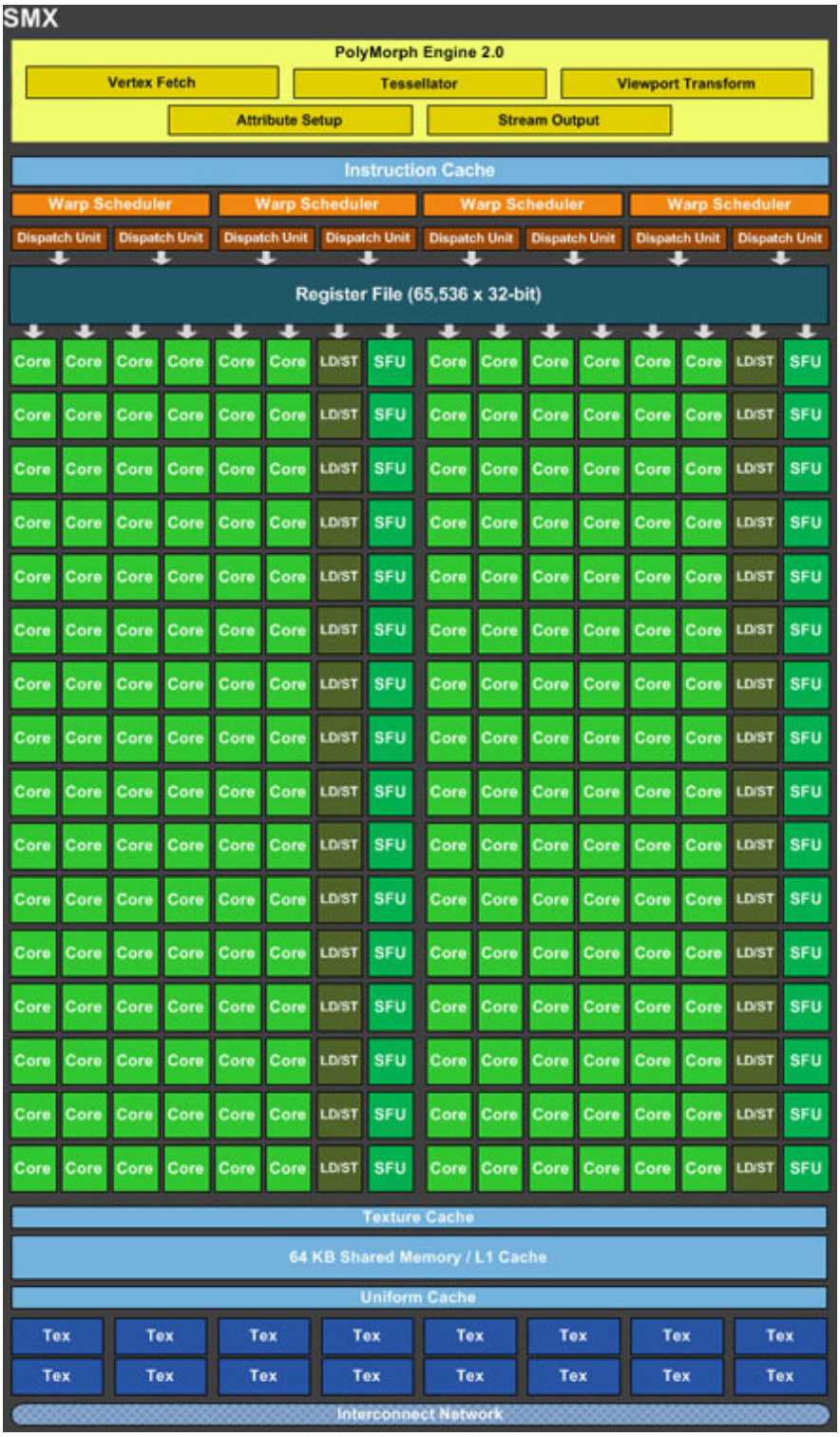


Figure 2.11: GeForce 680 GTX - SMX Diagram.

2.3 Logical graphics pipeline - DirectX 11 / OpenGL 4

The latest evolution of the logical graphics pipeline corresponds to DirectX 11 and OpenGL 4 specifications. Figure 2.12 shows the different steps of the pipeline according to the OpenGL 4 naming conventions. Among these steps it is important to distinguish fixed-function stages from programmable ones. Fixed-function stages are not modifiable while programmable stages may be completely designed by programmers. It is important to recall that each stage is processed in parallel onto the input data.

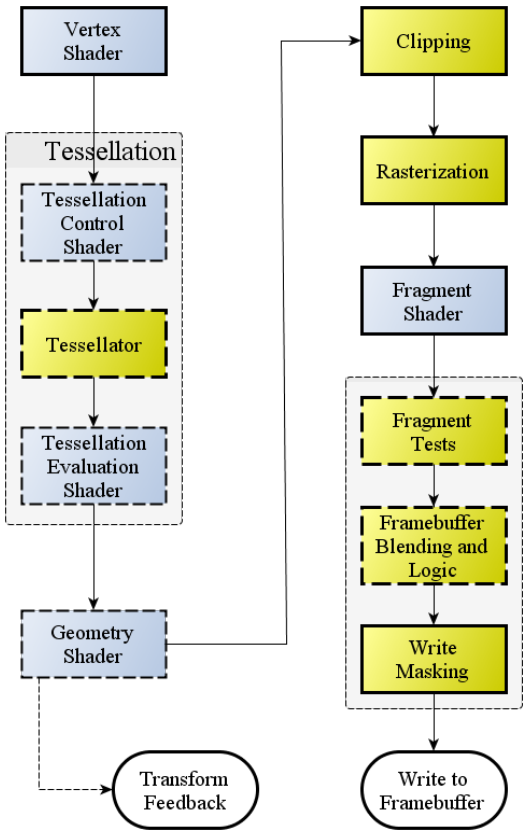


Figure 2.12: OpenGL 4.0 Pipeline.

- Vertex shader: This is the first stage of the graphics pipeline. Each vertex composing input primitives is sent to the vertex shader to be processed in parallel. Typically, per-vertex processing is vertex projection onto the camera space, normal transformation, preparation of additional information such as texture coordinates, etc.
- Tessellation control shader: At this stage, vertices are regrouped into patches. Tessellation control shader prepares the patch before actual tessellation. It indicates how should be the subdivision for the inner and outer borders of the patch.

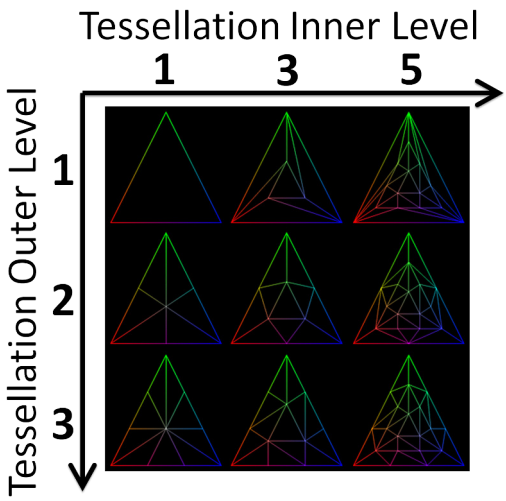


Figure 2.13: Triangle refinement according to the inner and outer tessellation factors.

- Tessellator/Primitive control: The hardware tessellation is really executed at this stage. Since it is a fixed-function stage, it automatically read the tessellation values filled at the tessellation control stage, and creates the new vertices resulting from the tessellation according to a tessellation pattern. Figure 2.13 shows refinement pattern on a triangle according to different tessellation factors.
- Tessellation evaluation shader: After tessellation, we need to specify some information about vertices issued by the primitive control stage, such as the position, the normal and the texture coordinates. In order to do this, primitive control stage informs on barycentric tessellation coordinates for each vertex. It allows the developer to know the position of the created vertex according to the boundary vertices of the patch. As he knows information about there boundary vertices, one may infer some values by interpolation using the barycentric tessellation coordinates.
- Geometry shader: In the geometry shader stage, one may compute per-primitive information, create again new vertex, or decide not to send them to the following stage (and not to display them). Input and output of this stage are specified primitives: points, lines, or triangles. At this stage, one may choose to record emitted vertices into a buffer for further usage (called transform feedback).
- Clipping/Culling: Before going through rasterization, we may easily discard non visible triangles. Primitives back-facing the camera may be eliminated. Primitives entirely outside the viewing region are discarded while those partly in the viewing region are clipped into fully visible and fully invisible parts to discard non visible ones.
- Rasterization: This is the step that convert primitives into fragments. Fragments selected as visible after fixed-function tests will constitute the pixels of the final image. Figure 2.14 shows an example of primitive rasterization. Generated fragments

are processed by the subsequent stages. Information like depth of the fragment are automatically computed and each fragment writes its depth on the depth buffer if its value is smaller than the previous one.

- Fragment shader: Each fragment is processed by a fragment shader. It computes some information such as the color, the texture applied, the lighting, etc. It may also recomputes its depth. Then some tests are run on the fragments before writing to the frame buffer.
- Fragment tests: scissor, stencil and depth tests are performed for each fragment. If at least one test fails, fragment is discarded. For instance, depth test compares depth of the current fragment to the depth of the fragment at the same frame buffer position. Only the closest fragment will be written in the frame buffer. For recent hardware, depth test can be computed before fragment shading in order to accelerate the process under certain conditions. This is know as early-z culling.
- Blending: If blending is required, color of the fragment at a given position will be blend with the color at the same position already in the frame buffer.
- Masks: Some masks can be defined to avoid fragments write the frame buffer depending on its color, depth or stencil values.
- Frame-buffer: Finally fragments that reach this stage are written to the frame buffer that composed the final image.

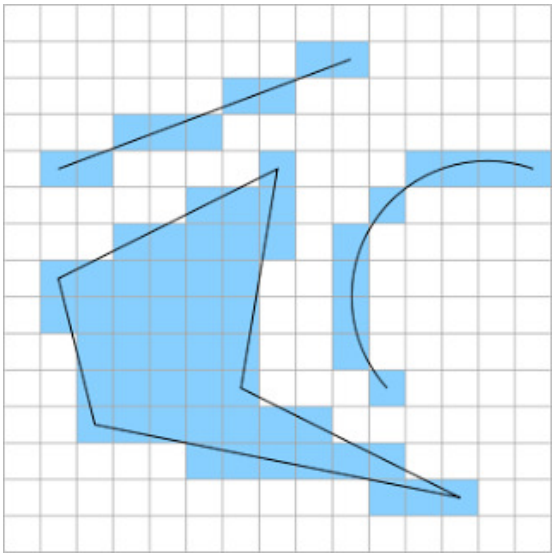


Figure 2.14: Rasterization converts polygons and lines (black lines) into fragments (blue pixels).

2.4 Conclusion

During the last decade, computing performances have been boosted thanks to the parallelism capabilities offered by recent affordable hardware such as multi-core CPUs and many-core GPUs. While CPUs are more efficient in computing complex operations on a single data set, GPUs are on the other hand more efficient for simple computations on very large number of data sets. In other words, powerful data parallelism may be achieved using GPUs for appropriate algorithms. Moreover, hardware tessellation recently introduced in GPUs allows for efficient real-time mesh refinement (DirectX11/OpenGL4). We believe that both of these advantages open new perspectives for high performance computing and we take benefit from them in this thesis. All our contributions from Chapter 2 to 7 systematically exploit these advantages in the solution designs.

In this thesis we aim at generating and rendering procedural objects efficiently using the GPU. The next chapter reviews some related work about the procedural generation and application on graphics hardware.

Chapter 3

Procedural content generation

In this chapter we address the procedural content generation, a very powerful tool for creating contents with infinite variations. We will start by introducing the idea of procedural content generation, the benefits and drawbacks associated to such a technique. Then we will see different methods specialized for creating contents such as vegetation and architecture. We will talk about artist-friendly methods and some research work focused on procedural geometry generation on surface. Finally we will see methods about real-time generation using GPUs.

3.1 Introduction to procedural generation of contents

Modeling an object may be done with various methods. Artists commonly use polygonal modeling softwares (3dsMax [Aut13a], Maya [Aut13b], etc) which offer many tools to create meshes. Traditionally artists manipulate primitives (points, lines, triangles, quads) in order to shape its model (Figure 3.1). Deformation operators (extrusion, split, displacement, twist, etc) help the artist to tune the model more easily than vertex by vertex. Other tools like Constructive Solid Geometry [RV77], FreeForm Deformation [SP86], and others assist the user to create and edit models in a more artist-friendly fashion. In order to help the artists, other softwares change the traditional polygonal modeling approach to a more artist-friendly way: the sculpting/painting. Zbrush [Pix13] and Mudbox [Aut13c] purpose to directly sculpt a base mesh to deform it and add details using brush tools. Other high-level methods for modeling exist such as subdivision surfaces. Artists may define only a coarse mesh using patch primitives. Subdivision surface methods such as Catmull-Clark [CC78] and Loop [Loo87] then refine this coarse mesh to a finer one Figure 3.2. Implicit surfaces are also used for defining a mesh as a mathematic function in $R3$ [Baj97]. Finally on top of all these methods, displacement mapping and other texture map based displacement methods, where vertices are moved

according to a value in a texture map, are trendy tools because it allows artists to paint the texture map directly on the object. Such painting techniques provide a high-level modeling mechanism to artists that avoid them to directly move vertices.

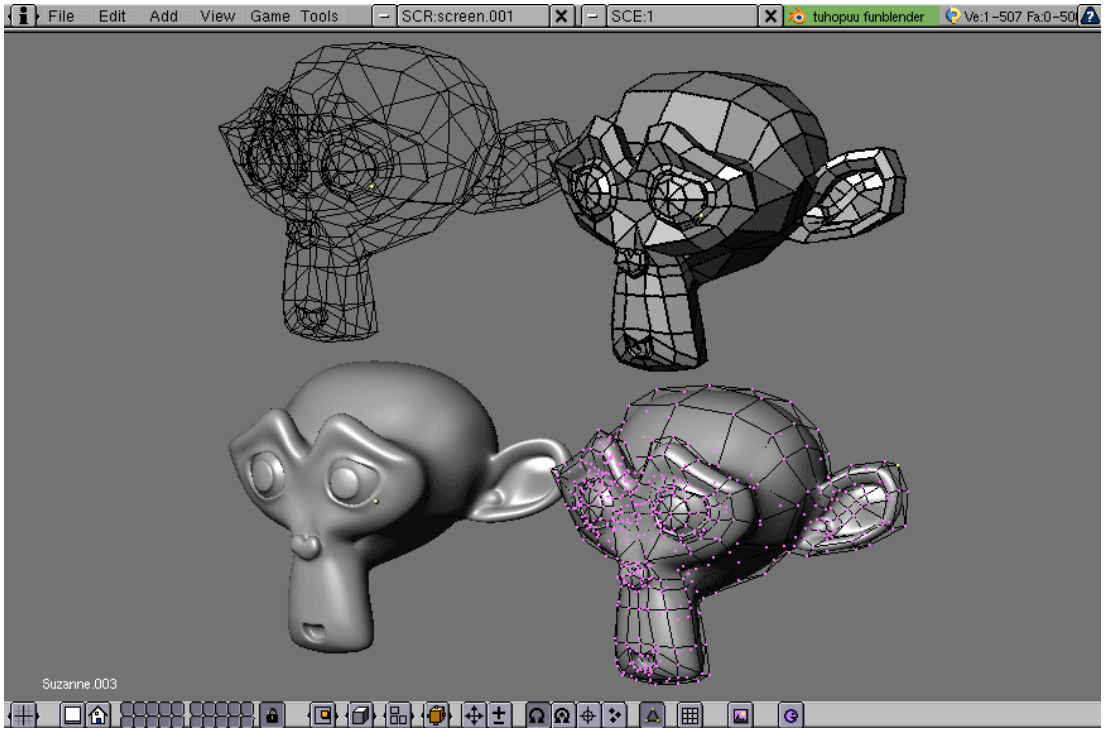


Figure 3.1: Mesh modeling using Blender software.

On the other hand procedural modeling approach completely differs from previous approaches. Instead of manipulating primitives, directly or through modeling tools, or sculpting the mesh, some objects may be described procedurally with an algorithm composed of a set of functions or rules calling each other. The main advantage of procedural modeling is the lightweight representation of the object. As we replace the actual polygonal mesh by functions describing its structure, we only need to store this small representation and not the final generated mesh. Another advantage is that a same set may generate potentially infinite number of similar but unique objects, when using stochastic and random functions. Fractals objects are procedurally generated. For instance, terrains with the diamond-square algorithm [FFC82].

Grammar-based methods also describe procedurally various objects with a succession of simple high-level grammar rules. Vegetation, architecture and terrains are among the most well-suited elements for grammar-based procedural modeling (Figure 3.3). For instance, one tree results from a growth phenomenon. It can be described with multiple growing rules, the first one modeling the trunk, and recursively smaller and smaller branches. A building facade is generally divided into two parts. A ground floor composed of doors and windows, a ledge being sit on top of the whole floor. Secondly,

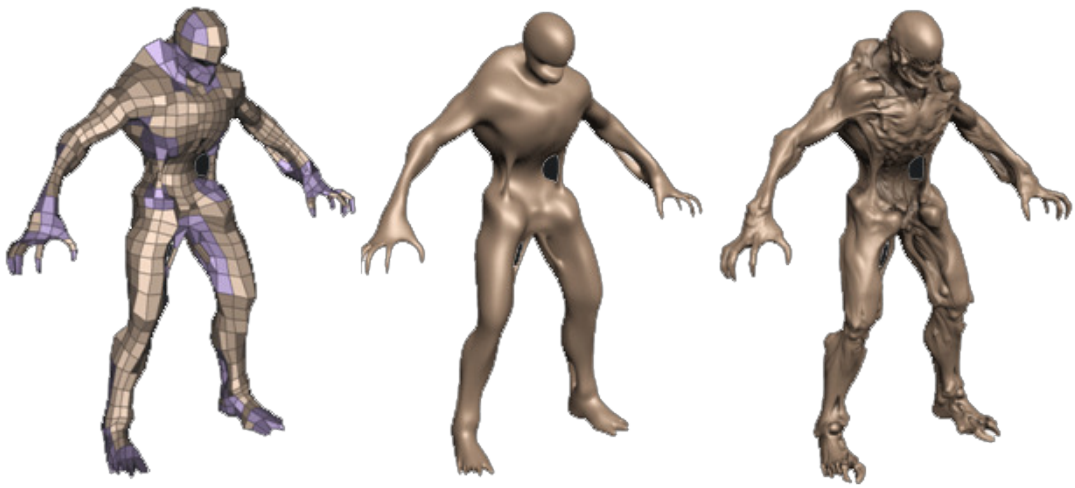


Figure 3.2: Highly detailed mesh modeled using subdivision surfaces with displacement mapping. Courtesy of Kenneth Scott, id Software 2008.

multiple stories composed of windows, balconies. Such a facade can easily be described with grammar rules.

Various objects and environments may be modeled using procedural techniques such as terrains ([SDKT+09, GGG+13, HGA+10]), road networks ([GG10, GPMG10]), urban layouts ([PM01, CEW+08, VKW+12]), vegetation ([DHL+98, BMJ+11]), architectures ([WWSR03, MWH+06, WOD09]), and interiors ([MSK10, LHP11]). Procedural methods have also been studied for synthesis of textures representing natural elements such as wood, bricks, concrete, tree bark, etc ([EMP+02, All13]). In this chapter we focus on grammar-based modeling of vegetation and architecture.

Figure 3.3: Vegetation and architecture are subjects of choice for procedural modeling.

3.2 Grammar-based modeling of plants

Procedural modeling was firstly introduced for plants generation, particularly using L-systems [Lin68]. We will describe the overall approach of such a method, however for readers who are interested in more explanation we recommend the book *The Algorithmic Beauty of Plants* authored by Prusinkiewicz and Lindenmayer [PLH+90].

A Lindenmayer System, also called L-System, is a string rewriting system originally developed as a mathematical theory of plant development. A string rewriting system is composed of a set of strings associated to rewriting rules (or production rules). The basic idea is to successively replace strings by the production rules to amplify the object. Starting from a simple string (the axiom), rewriting operations will replace it by a complex string, thus describing the shape of a complex plant. L-Systems are similar to Chomsky's grammars but differ in the way production rules are applied. Rewriting rules of Chomsky grammars are applied sequentially (one letter at a time), while L-System rewriting is processed in parallel (all letters at a time). This difference comes directly from the definition of the L-System, because simulating plant development means simulating cells subdivision where many divisions can occur at the same time.

Let us consider two strings, composed of only one letter each, a and b. We associate production rules $a \rightarrow ab$, and $b \rightarrow a$, meaning the letter a is replaced by the string ab, while the letter b is substituted by the letter a. Starting with the initial axiom string b, the rewriting process of the four first derivation steps is shown in Figure 3.4.

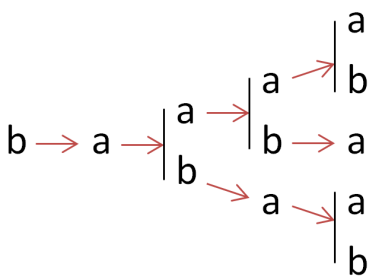


Figure 3.4: Rewriting process (5 derivation steps) according to the production rules $a \rightarrow ab$ and $b \rightarrow a$

In order to model plants, some graphical interpretations of L-Systems have been developed. Among them, the LOGO-style turtle interpretation is one the most known. The turtle interpretation is based on the turtle state (position, rotation) and some string commands understandable by the turtle. The turtle's state is defined as a triplet (x,y,α) representing the Cartesian coordinates (x,y) of the turtle and the angle *alpha* where the turtle is heading. Then, let us define d the step size when the turtle moves, and δ the angle increment when the turtle rotates. Given those two parameters, the following symbols are defined:

Finally, once the L-System is rewritten, we may draw it using the turtle interpretation where individual letters are treated as commands. For instance, considering the L-System defined by the axiom $F + F + F + F$ and the production rule

F	Move a step forward of length d, according to the current angle α, and draw the line
f	Same as F but without drawing a line
+	Rotate right the angle α by $+\delta$
-	Rotate left the angle α by -δ

Table 3.1: Symbols for turtle interpretation

$F \rightarrow F+F-F-FF+F+F-F$, Figure 3.5a) shows the interpretation of the axiom, while Figure 3.5b-c) correspond to $n = 1, 2$ steps of derivation of this L-System.

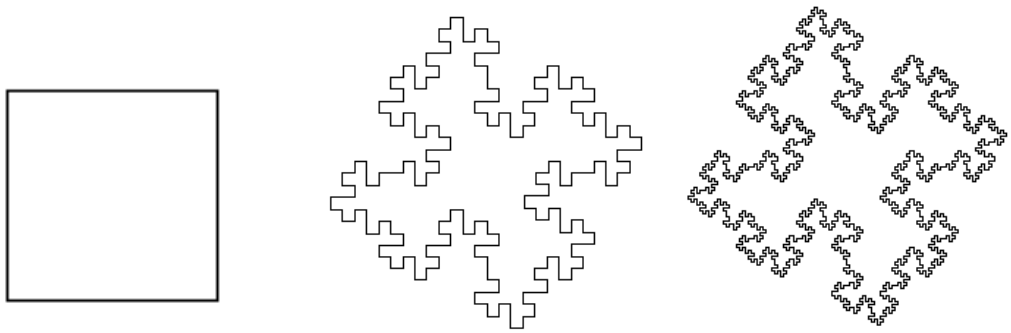


Figure 3.5: Generating a quadratic Koch island from a L-System.

On top of this turtle interpretation of L-Systems, other commands have been introduced. Without entering into details, bracketed L-Systems allow for describing branching structure of many plants by turtle state into a stack. Stochastic L-Systems, context-sensitive L-systems and other extensions have also been developed. Please refer to [PLH+90] for more details. Such L-Systems allow for modeling impressive vegetation as in Figure 3.6.

Due to their amplification role, L-systems have the main advantage to have a lightweight storage. Using only one L-System grammar and one seed, we may generate one highly detailed 3D model of a plant. Coupling with stochastic and context-sensitive parameters, many different plants can be generated with the same grammar. We only need to create random seeds to obtain a high range of diversity. However, such high definition models created with L-systems are costly to generate (derivation, interpretation) and render. While the generation and rendering of few models may be quite interactive, sceneries such as forest involving hundreds of trees are highly time-consuming and not interactive. Generation step may be avoided by storing amplified models but it implies a large memory cost. Some approaches try to optimize rendering of these plants by generating level of details, using the hierarchical structure of the object, to relieve the geometric load of the GPU [LCV03]. Other approaches accelerate generation time by parallelizing the derivation and interpretation steps [LH04, YHL+07, Mag09, LWW10]. A more detailed explanation will be given in the following section.

Figure 3.6: Different vegetation modeled from L-Systems, courtesy of Xfrog.

3.3 Grammar-based modeling of buildings

Procedural modeling has also been particularly useful for architecture modeling, especially for facades and buildings. We will describe briefly history of procedural architecture modeling, for more detailed information recent state-of-the-art reports are available [WMWF07] and [VAW+10].

While L-Systems have been designed for simulating growth phenomenon of plants, they have also been used for modeling buildings. Parish et al suggested to use L-System for architectural modeling [PM01]. They considered the starting axiom as the bounding box of the building, and then a building-like style L-System is derived on it. They also defined implicit level-of-details by stopping the derivation at an arbitrary level (see Figure 3.7).

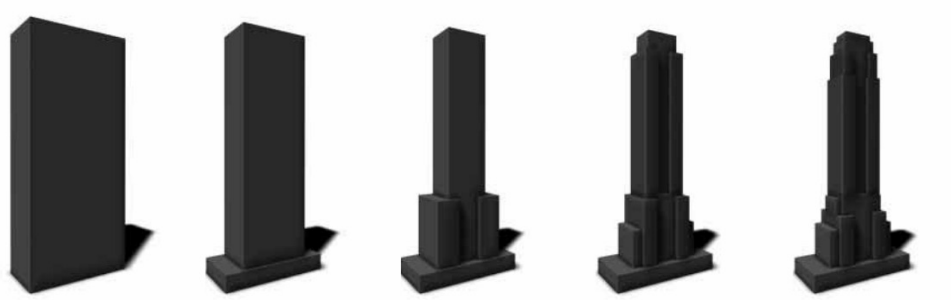


Figure 3.7: L-System for building. Five consecutive steps of the generation of a building, allowing easy LOD generation. Image from [PM01].

L-Systems have also been extended into FL-System (Functional L-System) to generate any kind of object hierarchy such as buildings [MPB05]. In this work, grammar symbols are replaced by functions that can be executed at any step of the rewriting process.

However, as a L-System simulates growth in open spaces, they are not really adapted to architecture modeling. On the other hand, Stiny *et al.* introduced shape grammars allowing to work on lines and points instead of string symbols [SG71] (see Figure 3.8). While L-System allows growth-like operations, shape grammars are more suitable for reduction-like operations. Shape grammars successfully address architecture design modeling [Fle87, SM+78, KE81, DF81, Dua02] but they are complicated to use and derivation of such grammars is a complex task, often needing a human intervention.

In order to simplify procedural architecture modeling, Wonka *et al.* introduced split grammars in Instant Architecture [WWSR03], an extension of shape grammars. In this work, authors defined a generic shape manipulation rule: *split* operation. It specifies how to split a shape into several smaller pieces, and then refine structures more and more (Figure 3.9). This split rule may take relative parameters as input so that a same splitting rule works on different shape sizes. The second operation rule is the *replace* rule, where terminal symbols may be associated to geometry shapes to achieve higher quality. Contrary to shape grammars, it provides a fully automatic derivation stage.

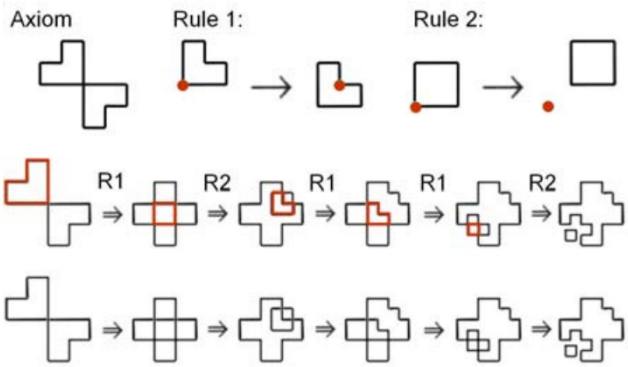


Figure 3.8: Shape grammar derivation starting from an axiom (top left) and two rules (top center and right).

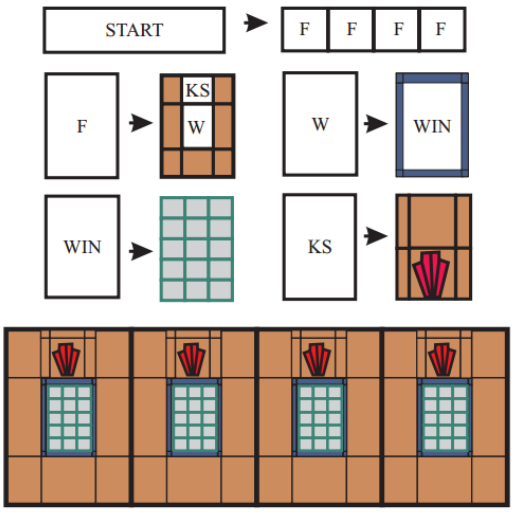


Figure 3.9: Split rules *start*, *f*, *w*, *win*, and *ks* are described on the top. The resulting split grammar derivation is shown on the bottom. Image from [WWSR03].

Muller *et al.* built upon previous split grammars a new grammar tool for architecture modeling: CGA shape grammar [MWH+06]. In their work, each shape has a scope associated representing its bounding box. They defined new generic rules such as *repeat*, *scale*, *translate*, etc. While dimension changing was not possible in split grammars, it is performed through the use of the new added *componentSplit* rule. Thus any volume may be subdivided at any time into faces, and any faces into segments. Formal description of CGA shape grammars will be detailed later in Chapter 4, as we base our work on the same grammar. They also handled snapping and occlusion thanks to grammar rules querying information to the surrounding environment. Figure 3.10 shows a set of rules and the building generated. CGA shape grammar method is integrated within CityEngine software [Esr12], where geometry shapes are modeled by hand using traditional tools.

```
PRIORITY 1:
1:  footprint ⇝ S(1r,building_height,1r) facades
      T(0,building_height,0) Roof("hipped",roof_angle){ roof }
PRIORITY 2:
2:  facades ⇝ Comp("sidefaces"){ facade }
3:  facade : Shape.visible("street")
      ⇝ Subdiv("X",1r,door_width*1.5){ tiles | entrance } : 0.5
      ⇝ Subdiv("X",door_width*1.5,1r){ entrance | tiles } : 0.5
4:  facade ⇝ tiles
5:  tiles ⇝ Repeat("X",window_spacing){ tile }
6:  tile ⇝ Subdiv("X",1r,window_width,1r){ wall |
      Subdiv("Y",2r,window_height,1r){ wall | window | wall } | wall }
7:  window : Scope.occ("noparent") != "none" ⇝ wall
8:  window ⇝ S(1r,1r,window_depth) I("win.obj")
9:  entrance ⇝ Subdiv("X",1r,door_width,1r){ wall |
      Subdiv("Y",door_height,1r){ door | wall } | wall }
10: door ⇝ S(1r,1r,door_depth) I("door.obj")
11: wall ⇝ I("wall.obj")
PRIORITY 3:
12: roof ⇝ Comp("sidefaces"){ covering }
      Comp("sideedges"){ roofedge } Comp("topedges"){ roofedge }
13: covering ⇝
      Repeat("XY",flatbrick_width,brick_length){ flatbrick }
      Subdiv("X",flatbrick_width,1r){ ε |
        Repeat("X",flatbrick_width){ roofedge } }
```

Figure 3.10: Houses on the right are generated using the grammar rule set on the left. Image from [MWH+06].

Krecklau *et al.* introduced Generalized Grammar [KPK10], a procedural modeling language adapting general modeling purpose to the power of shape grammars. Non-terminal classes such as boxes or FFD objects may be used to model complex object deformations. They also added abstract structure templates to facilitate parameter passing and object instantiation. They illustrated generalized grammar by modeling both buildings and trees Figure 3.11.

Interconnected structures such bridges and catenaries are quite challenging to model procedurally. Krecklau and Kobbelt specify attaching points on object that can be later used for interconnections [KK11a]. Respectfully to connection patterns or geometric queries, end points are selected among attaching points. Finally interconnection objects are either a rigid object chain constructed by inverse kinematics, or a deformable beam performed by spline interpolation (see Figure 3.11).

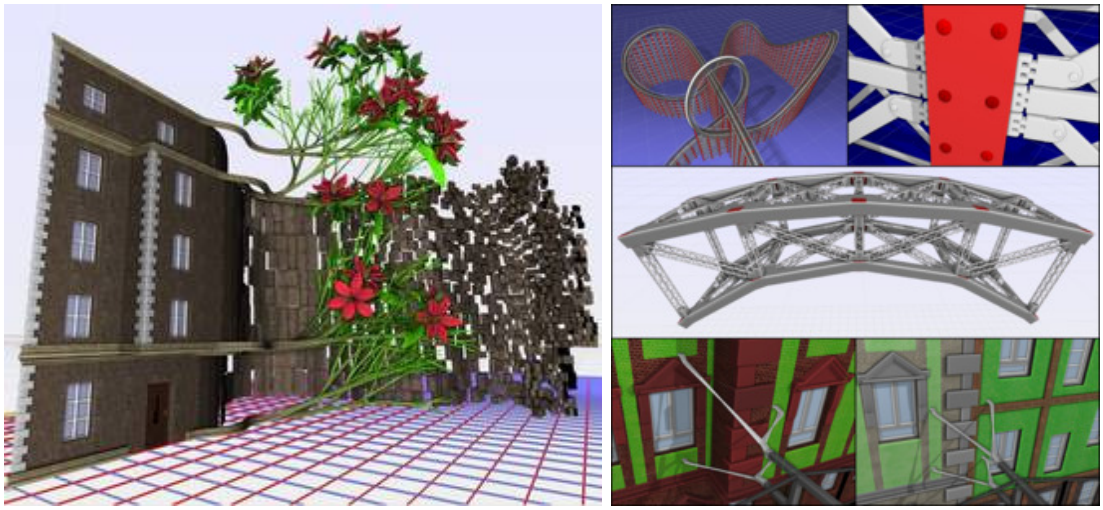


Figure 3.11: Generalized Grammar allows for modeling both vegetation and architecture (left, [KPK10]). Interconnection objects are designed by specifying attaching points (right, [KK11a]).

3.4 Geometry synthesis on surfaces

While grammar-based methods provide the ability to generate models freely in the 3D space, geometry synthesis on surface is challenging. Different methods for generating geometry on surfaces have been studied, often based on geometric textures. In these approaches, geometry to apply on the surface is defined as a 3D geometric texture and it relies on geometric texture synthesis. A work of Zhou *et al.* introduces mesh quilting [ZHW+06]. They defined a local pattern as a 3D texture sample to spread over the shell surrounding the surface mesh. The mesh should be parametrized and a graph cut minimization algorithm finds out how to assemble adjacent patterns according to the parametrization and local deformation, in order to seamlessly connect texture patches. To limit distortions on highly curved surfaces, they proposed a low distortion parametrization of the shell space. Compelling results are shown by the authors in Figure 3.12. However such method based on texture synthesis also has several limitations. It can only generate local or stationary patterns, meaning that only a similar pattern can be spread at a time. Then the way the pattern is spread over the surface is not user controllable at different places.

On the other hand, procedurally generated objects lying on surfaces are much challenging because they need to know information about the base surface mesh at any time. *Maya* software proposes a painting interface for creating procedural objects: *PaintEffects*. Only a few hard-coded objects are available. While an artist can edit some parameters, he cannot create its own grammar. During painting process, relative 3D position of the brush onto the mesh is found by ray casting. Then seeds are sampled along the stroke based on the input's speed (mouse or stylus). The quickier is paint the stroke, the farther apart are the seeds. Finally each seed is able to generate its own

Figure 3.12: A 3D sample of an ivy's stem and leaves is applied on the David mesh. With a vector field, the artist can design the final, complex mesh by guiding the mesh quilting algorithm. Here, the ivy is made to wrap around the leg and climb to the torso. Image from [ZHW+06].

geometry. As the generation is only guided by the surface normal at the initial seed position and does not follow the surface, intersections happen at concave surfaces. Such a method does not allow consistent global procedural generation where only one seed grows over the surface, replaced by many local generation.

Another software *An Ivy Generator* [Tho07] addresses the problem of consistent global procedural generation by relying on brute-force testing all the triangles of the scene [Tho07]. A seed is sampled on the base mesh, then a hard-coded grammar describing an ivy growth is processed. Basically for each growth iteration, it finds out the nearest triangle from the current scope to compute surface adhesion. This approach gives consistent results with limited intersections. However testing all triangles is a very heavy computation for complex scenes. Artists are not allowed to change the grammar or write their own, they may only edit the parameters.

In order to allow more artistic freedom, a recent work of Li *et al.* tends to combine geometry synthesis on surface with shape grammars [LBZ+11]. A user should define vector or tensor fields on the surface mesh to indicate the spread direction of the shape grammars onto the surface. Using shape grammars as geometry input for surface spreading allows global geometric pattern and not only local ones. Then they define shape grammar rules guided by those fields. Translation, rotation and scale rules may be driven by fields. The grammar can also use the field in the rule selection process itself. Compelling results are shown by authors (Figure 3.13). However even if designing vector or tensor fields on surface seems almost simple for artists, it remains difficult to have a good imagination of the results before generating the actual geometry. Moreover it necessitate a quite high generation time, preventing interactive edition.

Figure 3.13: A stem and leaf pattern growing on three different surfaces. Geometry structure is described by a shape grammar while growing direction are defined by vector or tensor fields. Image from [LBZ+11].

3.5 Improving artist experience

While procedural methods provide high-level modeling rules, one of the major drawbacks of such grammar-based methods is the lack of artistic control. Indeed, those text-based methods are not intuitive to artists, rules are often complicated to be easily used. Users have to learn how to construct well-defined objects, and most of the time they have no other choice than build grammars in a programmatic way. Moreover once a grammar is written, the artist should be able to tune the grammar parameters interactively, to avoid waiting the generated object. Actually artists have to spend much time and make much effort tuning parameters in order to get close from what they want or to an example. For instance, it is very stressful and time consuming to alter some branch positions of trees without changing the other branches. According to these two observations we may classify artist-friendly methods in two classes: grammar rules edition and grammar parameters tuning.

For methods focused on helping the artist to create and assemble grammar rules, Lipp *et al.* introduced an interface for interactive visual editing CGA shape grammars without writing any grammar rules [LWW08]. Such an approach try to combine procedural modeling to traditional visually-based modeling software. On the other hand, an interesting method from Kelly and Wonka presents a completely new approach for procedurally generated buildings, including roofs [KW11]. Contrary to previous methods that considered facades and roofs separately, they aim at modeling both facades and roofs together. Instead of using grammars for describing building structure, they use structure profiles which are artist-friendly tools indicating how a facade will be organized. Different profiles may be assigned to different parts of the building footprint, then a sweeping algorithm refines the building structure according to the profiles starting from the building footprint. Inspired by straight skeleton algorithm used for roof modeling, the sweeping process builds the extruded volume following the facade and roof profiles. Anchor points may be defined onto these profiles to add detailed shapes like windows, leading to complex buildings with compelling roofs as shown in Figure 3.14.

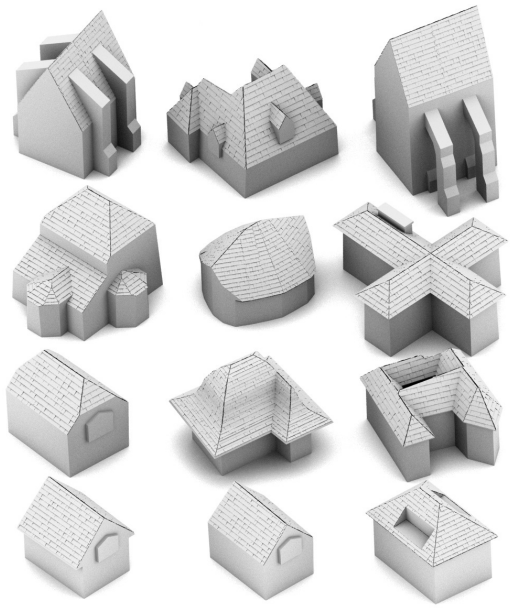


Figure 3.14: Describing buildings with extrusion profiles allows to modeled various style of architecture. Image from [KW11].

Second class of artist-helpful methods aim at simplify laborious task of grammar parameters tuning. In order to tackle this limitation, some approaches introduced external constraints for driving derivation thus simplifying artist control. Mech et Prusinkiewicz developed a modeling framework simulating interactions between plants as L-Systems and exterior environment constraints such as light and water competition, space colonization, collisions [MP96]. Prusinkiewicz *et al.* proposed the use of positional information to control parameters along a plant axis [PMKL01]. A sketching process for L-system modeling was introduced by Ijiri *et al.* where the artist defines a stroke along which the L-system will grow [IOI06].

Talton *et al.* recently proposed an algorithm for controlling grammar-based procedural methods [TLL+11]. The user may provide a high-level specification of the desired production either with geometric shapes, sketches, or analytical objectives (see Figure 3.15). Then the algorithm computes a production conformed to the specification by optimizing over the space of possible productions from the grammar. This is done by formulating procedural modeling tasks as probabilistic inference problems, with a costly RJMCMC (Resersible Jump Markov Chain Monte Carlo).

Another idea to easily tune grammar-based procedural models has been explored by Benes *et al.* [BŠMM11]. They built their framework on breaking the production system into a set of smaller ones, that may communicate with each other. The key idea is to divide the space into guides separating objects with closed boundaries, where each guide contains its own independent procedural model (Figure 3.16). Interaction between neighborhood procedural models is realized through the use of guide links serving as a message passing mechanism. Once a procedural model reaches a link, it

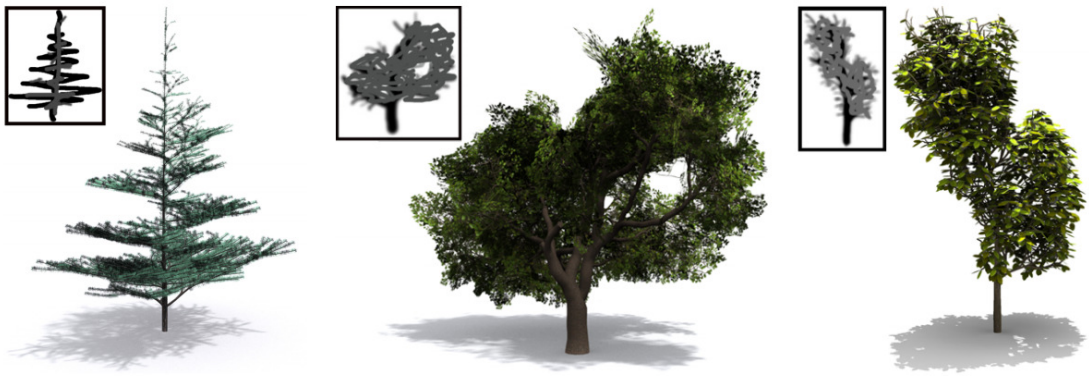


Figure 3.15: A simple sketch may constraint expansion process. Image from [TLL+11].

sends a message to the neighbor guide so that it may generate its own model. Artist control is possible either on the overall shape with the high-level specification guides, or on the local changes directly on the procedural systems.

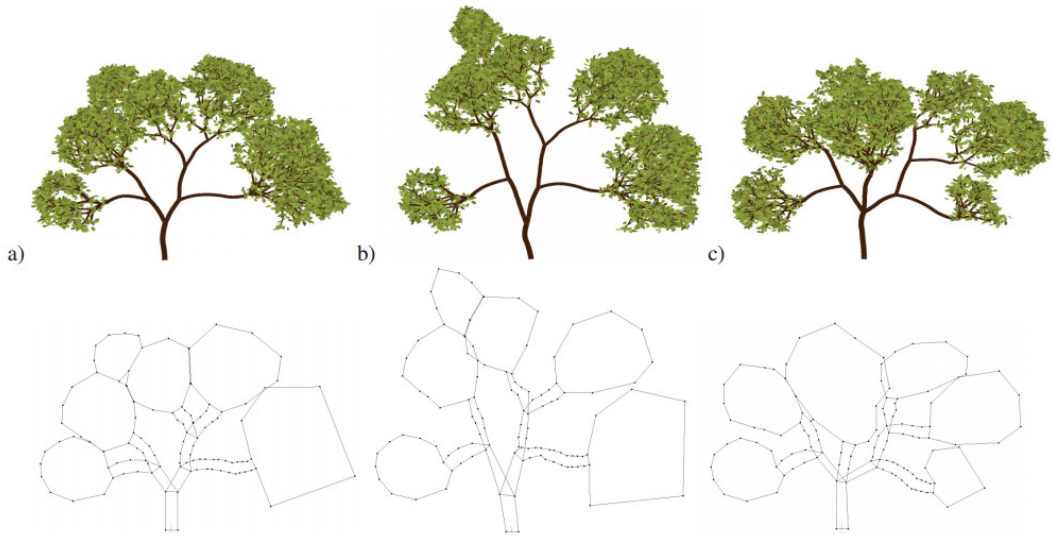


Figure 3.16: Guides are defined and linked to control the grammar expansion. Image from [BŠMM11].

Despite the powerful usage offered by procedural systems, artist experience is often depreciated because of the classical trial and error process in order to write manually the grammar generating the desired production. Some inverse procedural modeling methods aim at avoiding this tedious task by (semi-)automatically inferring the grammar rules and parameters based on an example models [ARB07, MZWVG07, BWS10, ŠBM+10]. As it is out of the scope of this thesis, we will not detail those methods however the reader should be aware that such methods exist to facilitate artist work. Grammars

extracted from inverse procedural system could be seen as input for our work.

3.6 Real-Time procedural generation on GPU

We presented some procedural methods in sections 3.2 and 3.3 that can, at the best, be generated at interactive time for relatively small scenes using the CPU. In case of potentially huge sceneries, derivation step of CGA shape grammars are order of magnitude too heavy to compute in real-time. Real-time rendering would necessitate generation of procedural models off-line for later rendering. However the amplification role of grammars automatically leads to a high memory cost if we store generated models. A real-time application would only have limited model size. This precludes real-time generation of massive cities for video games, and on-the-fly artist tuning of a building inside the massive environment. Allowing real-time generation of procedural contents would certainly open new gates for interactive applications industry.

In order to maintain a low generation time for the generation of massive facades, few methods tried to leverage parallelism capabilities offered by recent graphics hardware. Ali et al introduced a method for real-time rendering of building facades using the GPU [AYRW09]. Facades are compressed in an efficient manner and then rendering is performed in a two steps process. First, facades are decompressed on the GPU representing the lightweight facade structure. Then details are added through a ray-tracing algorithm to compute displacements using the fragment shader.

Direct grammar evaluation on the GPU has been proposed by Haegler *et al.* [HWA+10]. They proposed to store grammar rules as textures and to evaluate lazily grammars on-the-fly on the GPU. Grammar derivation is performed per-pixel using CUDA by a ray casting algorithm that only evaluate the corresponding branches of the rule graph. Krecklau *et al.* enhance [HWA+10] by handling stochastic rules and they added a interior mapping method [KK11b]. It allows for giving the illusion of interior modeling. While both methods provide interactive evaluation of facades and lightweight storage, they render only flat textured facades, and not detailed geometry shapes.

Marvie *et al.* introduced a ray-tracing renderer for procedural content [MGHS11]. Similarly to [HWA+10] grammar rules are encoded into textures and they perform a lazy ray-tracing on the GPU to evaluate the grammar (see Figure 3.17). Furthermore, their work is based on more general rules and geometry shapes as terminal symbols are supported. Using geometries encoded into images [GGH02], they compute the geometric facades elements again with ray-tracing in the same rendering pass when a ray reaches a leaf of the grammar. As everything is regenerated on-the-fly, it allows for interactive tuning of building parameters among massive datasets. While they obtained interactive compelling results on models with a low memory footprint, their method is heavily fragment-bound and not support component-split, occlusion and snapping rules.

While previous approaches address the problem of real-time generation of Chomsky based grammars, some other works focused on the parallel generation of L-Systems which are parallel rewriting systems. Lacz and Hart proposed a solution based on vertex and fragment shaders combined with a render-to-texture loop in order to compute L-

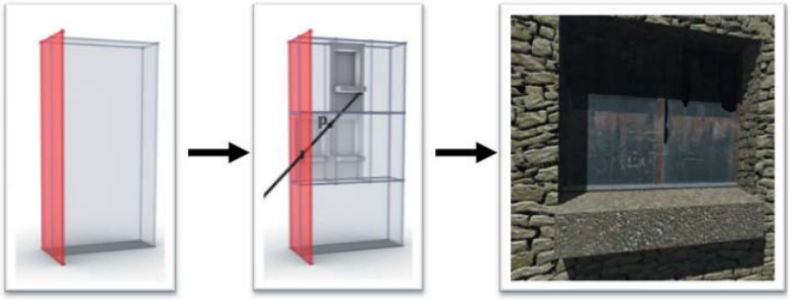


Figure 3.17: The grammar is evaluated on a per-pixel basis, and finally a ray-tracing of the terminal shape is performed. Image from [MGHS11].

System [LH04]. Madgics extends this concept using automatically created geometry shaders [Mag09]. Those two methods work only if successors have no effect on the traversal state which rarely happens and require a shader compilation step. Lipp *et al.* introduced a CUDA framework for the derivation and interpretation of L-systems using the GPU [LWW09]. They defined a GPU efficient structure representing a L-System. One iteration of the derivation step consists of three kernels running in order, each kernel is processed in parallel by thousands of threads. Kernels have to be launched as many times as desired producing an intermediate representation of the grammar for an arbitrary number of iterations. It extensively use the parallel scan primitive to achieve the computation. Interpretation step is also a three pass algorithm where the geometry is finally generated. For each branch of the L-System created, a new work item is prepared. Their framework allows for real-time L-system generation even for complex grammar (Figure 3.18). The major drawbacks of their method comes from the intermediate representation where a full regeneration of this structure is required when the iteration or any parameters change. They extended their work to multiple L-Systems [LWW10].

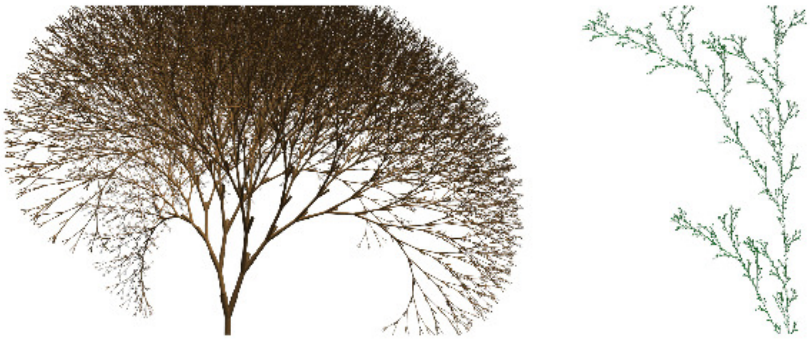


Figure 3.18: A multi-pass algorithm performs parallel derivation and interpretation of L-systems. Image from [LWW09].

3.7 Conclusion

Grammar-based procedural methods provides powerful modeling process for artists. Objects whose structure can be described by growth or reduction operations are particularly suited such as vegetation with L-Systems and architecture with CGA shape grammars. Objects are then lightweight decomposed in grammar rules representing the type or style of object, and grammar parameters driving those rules. Such decomposition allows for high variety by simply tuning the grammar parameters. However, two major drawbacks exist and motivate challenging research works. First, artist-friendly methods try to avoid direct manipulation of heavy text-based rules, and simplify interactive parameters tuning. Second, massive sceneries are limited because of both the high memory consumption and the non interactive generation.

Latest works take advantage of the parallelism capabilities of the GPU to reach interactive generation using fragment shaders. We believe the GPU may be efficiently used in order to allow interactive generation and tuning of massive sceneries. Chapter 4 develops our parallel approach based on a multi-stage GPU-based process, using intermediate caching to refine geometry at various levels and hardware tessellator of modern graphics hardware.

Chapter 4

Parallel procedural generation based on independent 1D atoms

In this chapter we introduce GPU shape grammars, a solution providing interactive procedural generation, tuning and visualization of the constitutive elements of environments for both video games and production rendering. Our technique generates highly detailed models without explicit final geometry storage. To this end we reformulate the grammar expansion to delay the generation of fully detailed models at the tessellation control and geometry shader stages. This reformulation to parallel segment-based expansion allows to substitute processing of complex input data for simple independent 1D atoms. We apply our solution to the interactive generation and rendering of buildings and trees. This work has been published in the GPU shape grammars paper [MBG+12].

4.1 Introduction

Modeling the massive, highly detailed environments used in current video games and cinema post-production requires intensive artistic input. The geometry of those environments can be handled in real-time by game engines, to the extent of the available graphics memory depending on the geometric complexity. In the context of production rendering, the creativity is often restrained by the lack of appropriate real-time feedback. The elements of complex scenes are often modeled independently, while the assembly is performed on very low-resolution models to preserve interactivity. Once assembled the memory footprint of the scene commonly exceeds the memory available on commodity hardware, hence requiring specific pre-processing and out-of-core schemes for visualization.

The amount of authoring efforts can be drastically reduced by procedural modeling, which exploits the repetitive patterns typically present in buildings, cities and organic shapes. Instead of explicitly modeling the scene elements, the artist selects a procedure describing the construction rules for the object or family of objects. Using a collection of elementary (possibly high definition) shapes provided by the artist, the actual geometry of entire families of objects is then generated automatically, sparing the user from modeling the entire object structure. However, modifying a single parameter of the construction rules may require regenerating the entire object, resulting in a loss of interactivity. This issue scales with the size of the generated environments: large sceneries such as virtual cities comprise many objects generated from a small set of construction rules. Modifying a rule then involves a full generation and storage of all the related models (Figure 4.1), resulting in delays in the design work-flow. Furthermore, the memory occupancy of a large set of fully detailed objects quickly rises to prohibitive levels.

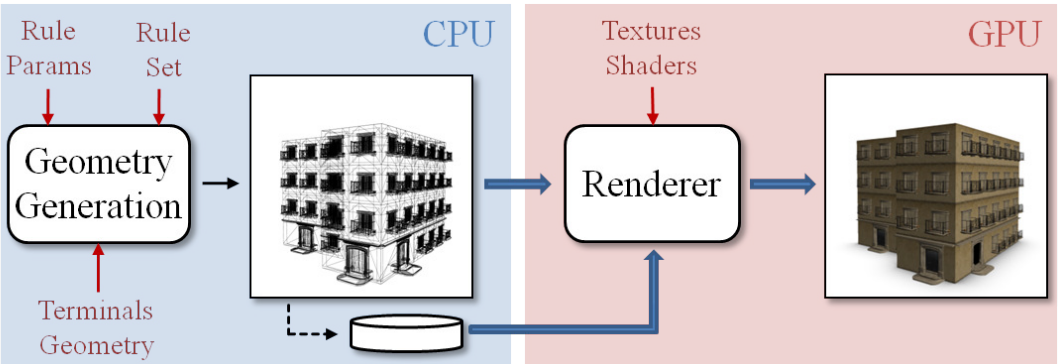


Figure 4.1: Classic generation pipeline process the grammar and generates models using the CPU and then transfers the result to the GPU for rendering.

We introduce a new method for real-time modeling and visualization of complex environment elements using procedural modeling. In order to benefit from the parallelism capabilities of the GPUs, we propose a parallel segment-based grammar expansion work-

ing on independent simple 1D atoms, corresponding to the decomposition of complex input data to elementary elements. Then, our method expands construction rules on the fly taking advantage of the tessellation units of modern graphics hardware. We propose a complete pipeline for the generation and rendering of procedural models, avoiding the explicit storage of the scene geometry (Figure 4.2). Our solution is easily integrable within existing rendering workflows, thus brings the advantages of procedural modeling within real-time rendering engines, for which computational and memory efficiencies are mandatory.

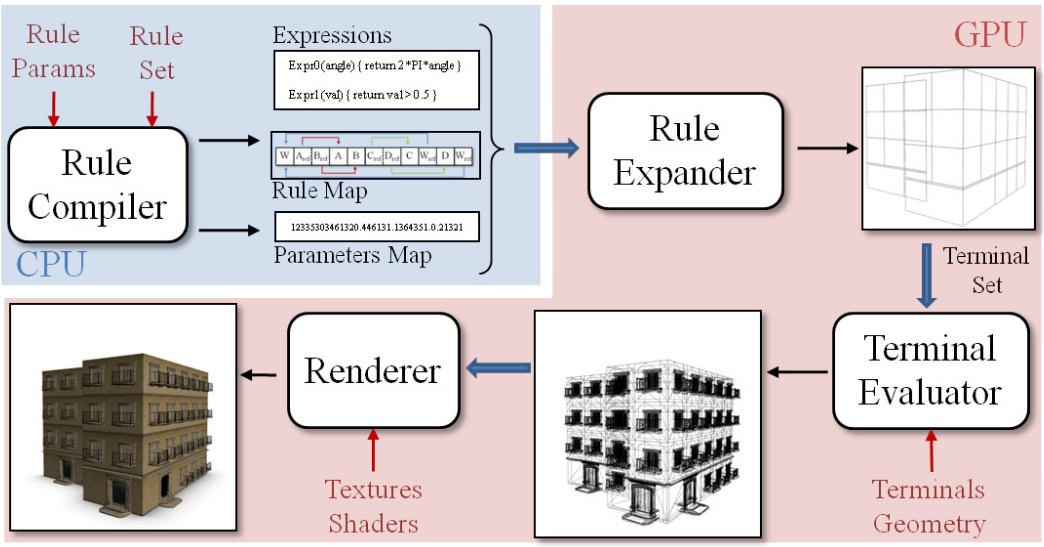


Figure 4.2: GPU shape grammars map procedural modeling techniques to the requirements of graphics hardware. We compile the grammars into an efficient structure for fast expansion of the rules. The expansion generates a lightweight intermediate representation of the object structures. The geometry is then generated on the fly without explicit storage.

In Section 4.2 we introduce *GPU shape grammars* and discuss a generic pipelined architecture for the generation of highly detailed procedural models (Sections 4.3 to 4.5). We apply our approach to the generation of buildings and vegetation, and we show parallelization results (Section 4.6).

4.2 Procedural pipeline overview

Procedural generation techniques are usually based on iterative refinement of a data structure, while the efficiency of the graphics pipeline comes from a highly parallel, stage-based structure carrying specialized information. We then reformulate grammar expansion for efficient procedural generation on graphics hardware. Based on Chomsky grammars [Cho65], our solution supports both growth and reduction operations and is usable for most purposes of procedural modeling. As shown in Figure 4.2 our approach

is divided into three main components: the rule compiler, rule expander and terminal evaluator.

Grammars represent high level mechanisms while GPU shaders support relatively low level operations. In Section 4.3, our CPU-based compiler extracts a generic *rule graph* from the rules (*i.e.*, organization of the rules with each other) and convert this graph into a GPU interpretable *rule map*. The run-time behavior of the rules is extracted into shader expressions and combined with a generic *rule map* interpreter, yielding a grammar-specific GPU *expression-instantiated interpreter*. Finally, the grammar parameters are packed into a GPU-compatible *parameter map* for fast access.

The rule expander runs our interpreter on graphics hardware to traverse the rule map according to the generation parameters (Section 4.4). For each node, the expression-instantiated interpreter evaluates the corresponding expression and performs a depth-first traversal. The output of the traversal is a lightweight set of terminal symbols only describing the object structure. The actual geometry is not generated at this stage for high speed processing and low memory usage.

Finally, in Section 4.5 the terminal evaluator performs GPU-based geometry generation based on this terminal set. The evaluator fetches the geometric description of each terminal and generates the terminal geometry at the desired location. The geometry is then directly rendered, avoiding any storage of the fully detailed models.

4.3 Rule compiler

The first stage of the GPU shape grammars pipeline is the CPU-based rule compiler which build a rule map from the grammar rules (see Figure 4.3). A grammar rule can be generically written in the spirit of [MWH+06]:

$$\text{Pred} \rightsquigarrow \text{Rule}(\{Expr_j(P)\}_{j\in\mathbb{N}})\{\text{Succ}\} \tag{4.1}$$

where *Pred* is the predecessor of the rule, *Rule* is the name of one of the supported built-in rules, and *Succ* is the set of successors of the rule. The behavior of the expressions $Expr_j$ is driven by the generation parameters $P = \{p_i\}_{i\in\mathbb{N}}$. Note that compared to [MWH+06], we replace the rule condition by the specific rule type *Condition* which switch between two successors according to the evaluation of the expression.

For example, let us consider a grammar simulating a simple recursive growth (Figure 4.4a). In Figure 4.4b we express this grammar in terms of the components of the generic rule (Equation 4.1).

Using this formulation we represent a rule using four components: a predecessor, a successor set, a rule type and a set of expressions representing the related arguments. All the possible expansions of a grammar can then be represented by a *rule graph* linking each predecessor to all its successors through a rule type identifier and a set of expressions associated to each rule (see Figure 4.5a).

We flatten the rule graph into a rule map (Figure 4.5b). For each rule the successors are represented using offsets within the rule map, while the corresponding rule type

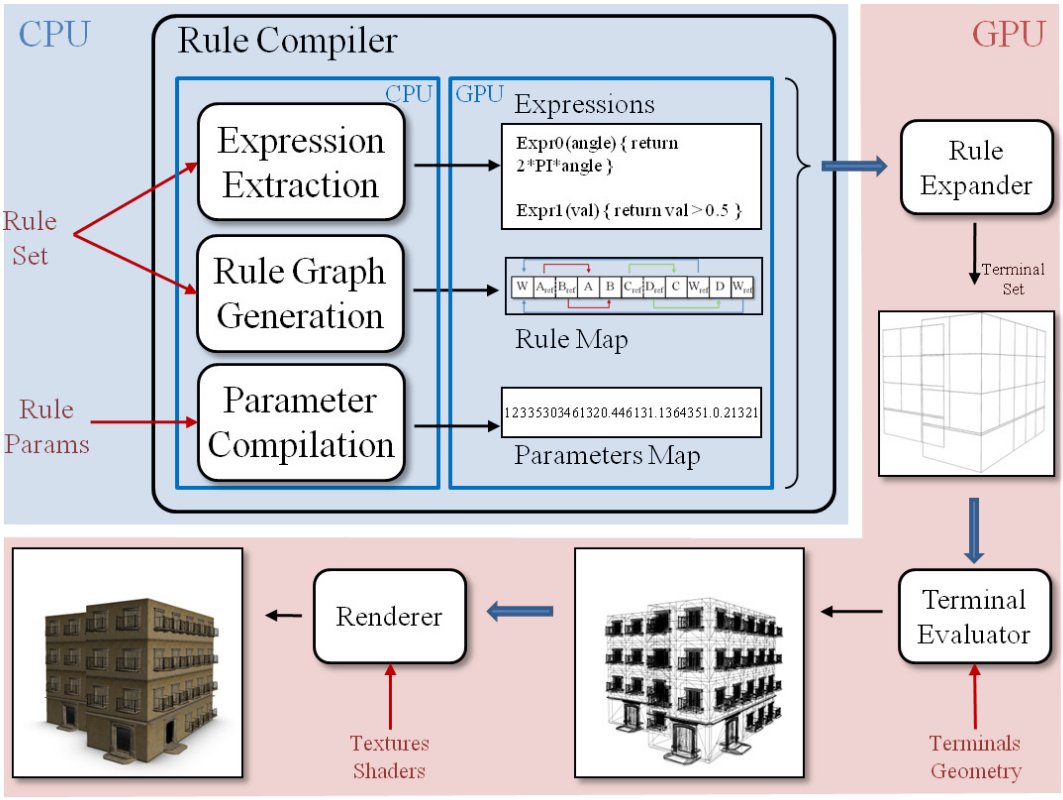


Figure 4.3: The rule compiler divides the grammar into a rule map and a set of expressions expressed in shader code. The expressions are combined with a generic rule map interpreter for fast grammar expansion on graphics hardware.

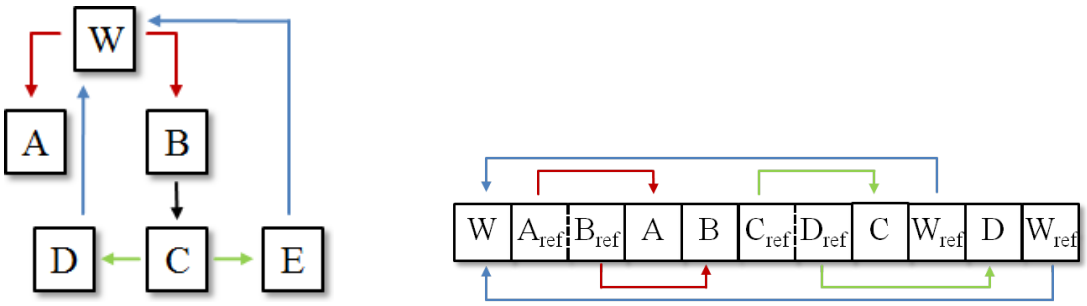
W $\rightsquigarrow$ Extrude(10){A,B}
A $\rightsquigarrow$ Shape(shapeId)
B $\rightsquigarrow$ Cond(recLevel<n){C,$\emptyset$}
C $\rightsquigarrow$ Branch{D, E}
D $\rightsquigarrow$ Rotate($-\pi/4$){W}
E $\rightsquigarrow$ Rotate($\pi/4$){W}

(a) Expansion grammar

Pred	Rule	*Expr*	Succ
W	Extrude	10	{A,B}
A	Shape	shapeId	
B	Cond	recLevel<n	{C,$\emptyset$}
C	Branch		{D, E}
D	Rotate	$-\pi/4$	{W}
E	Rotate	$\pi/4$	{W}

(b) Generic rule components

Figure 4.4: A simple growth grammar (a), where W is the axiom. The components of the generic rules are specified in (b).



(a) Rule graph (b) Rule map

Figure 4.5: Each component of the rules is encoded into a *rule graph* (a). This graph is flatten into a GPU interpretable *rule map* b).

is replaced by a simple identifier. The argument expressions typically involve low-level operations compatible with standard shader languages. Using this observation we extract the expressions from the grammar and generate an expression library in shader code for run-time evaluation, allowing on-the-fly evaluation of the expressions.

This representation accounts for any possible expansion of the input grammar. The automatically generated shader code only describes the expressions related to each rule individually and does not implement the interpretation of the rule map. We then introduce a stack-based generic rule map interpreter, performed at rule expansion stage.

4.4 Rule expander

The grammar and generation parameters fully describe the structure of the target object, although not in a directly renderable form. In order to achieve efficient grammar interpretation on GPUs, the rule expander makes intensive use of the geometry generation capabilities of graphics hardware to create a set of simple primitives associated to each terminal symbol (Figure 4.6) and we introduce a *segment-based expansion* based on a grammar-specific expression-instanciated interpreter.

4.4.1 Grammar-specific GPU expression-instantiated interpreter

The rule expander is a GPU stage responsible for interpreting the grammar by traversing the rule map and evaluating the expressions. We take advantage of the separation of the grammar into a generic rule map and shader code expressions to introduce an independent GPU interpreter. Our interpreter is designed to handle any grammar rule set so that we need to compile it only one time. Grammar-specific expressions are regrouped together into a library and compiled separately from the generic interpreter. Then, to evaluate the expressions at run-time, the interpreter reads the expression id from the rule map and asks the library to compute the expression corresponding to this id. It results in a black box expression computation. In summary, the rule expander proceeds as follows: for each rule, the interpreter traverses the rule map, evaluates the

expressions blinded and applies the corresponding rule to determine the successor set. The successors are then recursively processed until all the rules have been expanded into terminal symbols (Algorithm 1).

The combination of a generic interpreter with the user-defined grammar expressions could be performed using dynamic libraries or runtime assembly code generation on a classical processor. As current graphics hardware does not allow the use of such libraries we automatically combine the generic interpreter code with the grammar-specific expression library. The result is a high performance, expression-instantiated interpreter for the target grammar ready to be executed at any stage of the graphics pipeline. While we designed an efficient GPU grammar interpreter, performing an expansion on a complex input data generally does not fit well the graphics pipeline. Because GPUs prefer working on multiple instances of a simple data, we thus introduce a segment-based expansion, preventing complex operations.

4.4.2 Segment-based expansion

A reason for the high performance of graphics hardware is the subdivision of a multi-dimensional problem into problems of fixed dimension. For example the vertex proces-

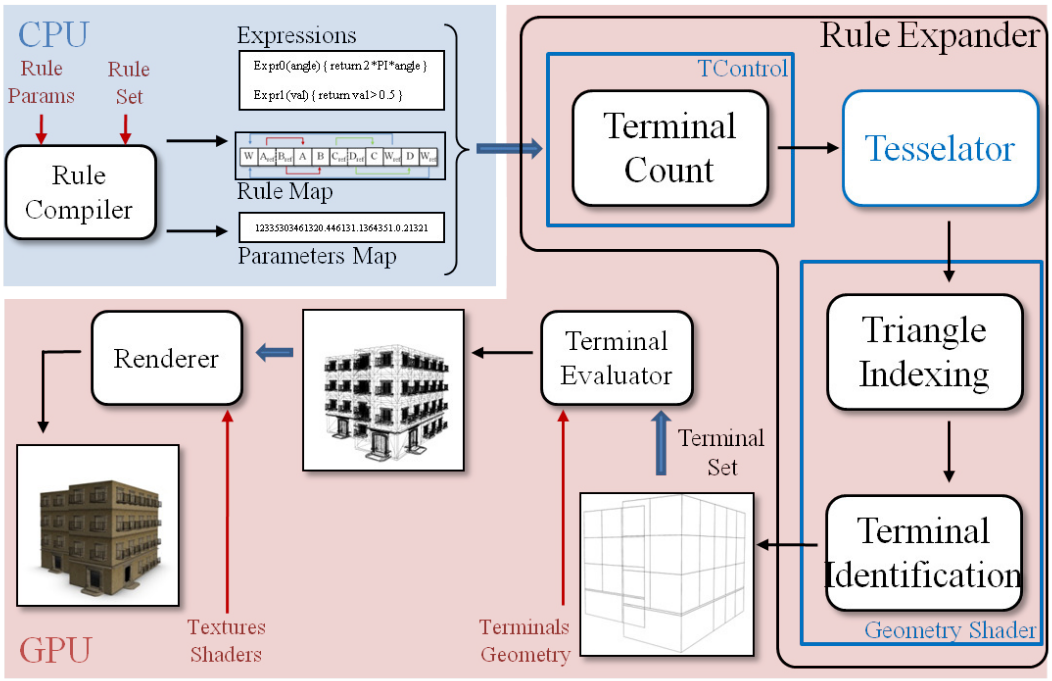


Figure 4.6: The rule expander interprets the grammar according to the generation parameters, generating a terminal list and one triangle per terminal. Then, the geometry shader associates each triangle with a terminal, and outputs a lightweight structure representing the terminal symbols.

```
ruleStack.push(Axiom)
while !ruleStack.empty() do
  curId = ruleStack.pop()
  if curId is a terminal then
    terminal = RuleEval( curId )
    terminalList.add( terminal )
  else
    succ = RuleEval( curId )
    ruleStack.push( succ )
  end if
end while
```

ALG 1: Generic rule map interpreter algorithm. Once the interpreter meets a terminal symbol, it stores the corresponding terminal for further evaluation (Section 4.5). For each non-terminal symbols, we evaluate the corresponding rule and push the successor(s) into the stack. Function *RuleEval()* evaluates the grammar rules and, if expressions have to be evaluated at run-time, it asks the expression library to compute it according to the expression id.

sor only considers points, while the geometry processor is applied to primitives. The pipeline for GPU shape grammars follows a similar idea: optimization opportunities arise when formulating the grammar expansion scheme in terms of atomic elements of fixed dimension.

The operations described in [MWH+06] operate on elements of various dimensions from points to volumes, where the *Component Split* operation breaks elements into elements of smaller dimension. While this approach is efficiently implementable on a general processor, the underlying data-parallel structure of graphics hardware cannot handle the dimension changes without a significant overhead. In counterpart, this architecture favors repetitive operations performed in parallel on objects of constant dimension.

This problem is naturally solved by a simple principle: an object of dimension n can always be decomposed into elements of lower dimensions from 0 to n [Edm60, Lie94]. As 3D surfacic objects are composed of elements of dimension 0 (vertices), 1 (segments) and 2 (surfaces), geometric operations can be conveniently expressed as a combinations of 1-dimensional elements with potential 2D elements. For example, the extrusion of a surface is a decomposition of the surface into a set of segments, followed by an extrusion of each segment and a translation of the initial surface. A splitting operation on the resulting surface can be performed by subdividing the surface into two sets of segments. Following this principle, we introduce a GPU-compliant grammar expansion method based on 1D atoms.

This formulation has a direct impact on the representation of the rules: even when applied on 1D atoms, a notion of the higher dimensions must remain temporarily to preserve the power of expression. Considering a single input atom (Figure 4.7a), the

(a) 1D atom case (b) Expansion element case

Figure 4.7: Starting from a single 1D atom (a), our segment-based formulation generates an *expansion element* (*i.e.*, a 2D atom and a set of contour 1D atoms) and the initial 1D atom is translated. We may also start from such an expansion element (b): each contour 1D atom is segment-based expanded as in (a) and the 2D atom is translated.

output of this rule is twofold: First, the generated face is described by a 2D atom and a set of contour 1D atoms, called *expansion element*. Second, the rule translates the input 1D atom by the extrusion vector $\mathbf{v}$. More formally, this results in the following syntax for the extrusion:

$$\text{Pred} \rightsquigarrow \text{Extrude}(\mathbf{v})\{\text{F_Succ, S_Succ}\} \tag{4.2}$$

where *F_Succ* and *S_Succ* are respectively applied to the generated expansion element and to the translated atom. While we illustrated the *Extrude* rule starting from a segment, this principle straightforwardly generalizes to the expansion element (Figure 4.7b). In this case, the contour 1D atoms perform their own segment-based expansion as presented, and we also translate the starting expansion element at the top. Thus, each initial contour 1D atoms is extruded to an expansion element that may be also used for futher extrusion. Finally, *F_Succ* is always applied to an expansion element while *S_Succ* may be applied either to the translated 1D atom or to the translated expansion element.

Each rule expander thread uses a context representing the local frame of the current atom and a tag indicating whether the atom is part of a surface. Current contexts are managed using fixed-size stacks in GPU memory. Starting with an input segment, we initialize an *expansion context* representing the frame of the segment. The axiom W of the grammar is then applied, generating a set of 1D and 2D atoms and pushing the associated local frames into the expansion context. The successors of W are then applied recursively until all paths reach terminal rules. We also provide current-scope and world-scope accessors within the grammar. Those accessors can be used as parameters for any rule. Initial expansion context is accessible through world-scope accessors, while the current expansion context is questionable with current-scope accessors. For instance, one may query the initial normal of the 1D atom (corresponding to the normal of the footprint) with *getWorldNormal()*, or the normal of the current atom after multiple extrusions using *getCurrentNormal()*.

In Figure 4.8, we illustrate the segment-based expansion onto the grammar example in Figure 4.4, generating a tree-like structure. We recursively apply expansion on 1D

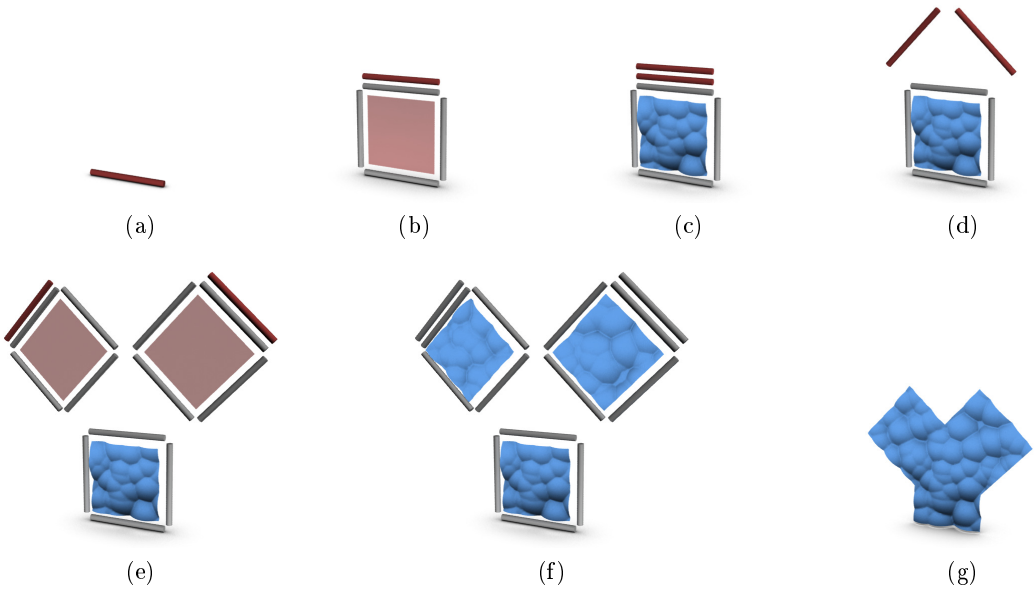


Figure 4.8: Exploded view of the expansion of our example grammar using segment-based formulation. The axiom is applied on a base 1D atom (a), yielding an extruded face composed of one 2D and four 1D atoms (*i.e.*, an expansion element), and a translated 1D atom (b). The expansion element is replaced by a terminal shape, while the translated 1D atom is divided into branches (c) and rotated (d). The branches are then extruded (e) and the new expansion elements are replaced by shapes (f), resulting in the final generated geometry (g).

atoms and we associate terminal shapes to the expansion element. As stated into the grammar, once we reach a given recursion level we stop the expansion by redirecting the condition rule to its second successor which is a *null terminal* ($\oslash$ notation). Each time we hit a null terminal symbol during grammar expansion, we interrupt the evaluation of the current branch.

Our implementation of the interpreter supports a subset of the rule types introduced in [MWH+06], including split, repetition and extrusion. As we focus on massively parallel generation, we did not address the rules involving a dependency between the generated objects, such as the occlusion-related operations. The consequences of the limitations of our method are discussed in Section 4.6.3. The Appendix A shows the rules implemented within GPU shape grammars, according to the segment-based formulation.

In addition to the rules presented in Appendix A, we introduce a *branch* rule which simply replicates the original atom n times (Figure 4.8c) and applies a successor to each

generated segment as described in Figure 4.4:

$$\text{Pred} \rightsquigarrow \text{Branch}(n)\,\{\ \{\text{Succ}_\text{i}\ \}_{i=1\ldots n}\} \tag{4.3}$$

This segment-based approach provides a solution for efficient grammar expansion on graphics hardware based on 1D atoms and a subsequent reformulation of the classical rule syntax. Our reformulation is carried out automatically from the input grammar, making this process logical to the grammar designer. In the remainder of this section we introduce the mapping of this solution within the graphics pipeline.

4.4.3 Implementation on graphics hardware

In order to efficiently generate the terminal sets, the rule expanding stage takes advantage of the geometry generation capabilities of GPUs. We implemented our technique using hardware tessellation and Shader Model 4.0 (DirectX 11, OpenGL 4.0). While our technique could also be implemented using Cuda/OpenCL languages, to our knowledge the tessellation units are separated from the computing cores and cannot be directly accessed. An implementation using the graphics pipeline then allows us to harness the computational power of both the computation cores and tessellation units.

Following the tessellation scheme of graphics hardware our approach is divided into two parts: the generation of the terminal set of the object at the tessellation control shader stage, and the output of the corresponding terminal primitives at the geometry shader stage. Figure 4.6 shows the rule expansion steps according to the graphics pipeline.

Starting from input 1D atoms, the tessellation control shader executes our expression-instantiated rule interpreter. The output of the interpreter is then a list of terminal symbols with their associated parameters stored within a simple read/write buffer in graphics memory (using shader image load store mechanisms). As the hardware tessellator can not generate the exact number of triangles corresponding to the list size, we have to optimize the generation step. For a quadrilateral patch, specifying i and j tessellation levels forces the hardware tessellator to generate $i \times j$ quadrilaterals, each one being subdivided into 2 triangles, yielding to $2 \times i \times j$ triangles. In order to minimize the number of unnecessary triangles generated, we compute the minimum i and j levels such that we generate at least the list size. Then we specify tessellation levels i and j to the tessellator which generates the appropriate number of triangles.

Once the triangles have been generated by the hardware tessellator, we consider the positioning of the terminal set stored into the previously generated terminal list. The idea is to replace each generated triangle by a terminal shapes. However, the output of the tessellator is a set of triangles only identified by their barycentric coordinates. In the geometry shader, we first generate a unique identifier for each triangle based on those coordinates. The algorithm 2 describes how we arbitrarily compute a per-triangle index for a quadrilateral depending on the barycentric tessellation coordinates. An example of tessellation pattern hardware-defined and its computed per-triangle indexing are shown in Figure 4.9. Using this identifier, we fetch the corresponding terminal symbol and parameters from the previously generated list and associate this information to the

triangle. Triangles whose identifiers exceed the list size are simply discarded. The final output of the geometry shader is a lightweight set of primitives representing the placement and parameters of the terminals. This terminal set can then be streamed to the next step of our pipeline for direct geometry generation and rendering. If the generation parameters are constant, the set can also be cached to avoid per-frame regeneration. As the terminal set does not embed the geometry of the terminals, this caching incurs a very low memory overhead. The set of terminals is then passed to the terminal evaluator for geometry generation.

Compute the minimal tessellation coordinates (u_{min}, v_{min}) of the triangle.
Those coordinates identify the quadrilateral containing the triangle.
Determine if the triangle defines the upper or lower part of the quadrilateral
if An edge of the triangle is along v_{min} **then**
  The triangle defines the lower part of the quadrilateral
else
  The triangle defines the upper part
end if
Compute an indexation base, $base_u = u_{min}$
if The triangle defines the lower quadrilateral part **then**
  $base_v = v_{min}$
else
  $base_v = v_{min} + \frac{1}{2j}$
end if
Compute the triangle index: $index = base_u \times i + 2 \times base_v \times i \times j$

Alg 2: Triangle indexing for a quadrilateral based on barycentric tessellation coordinates u, v, according to the tessellation levels i and j.

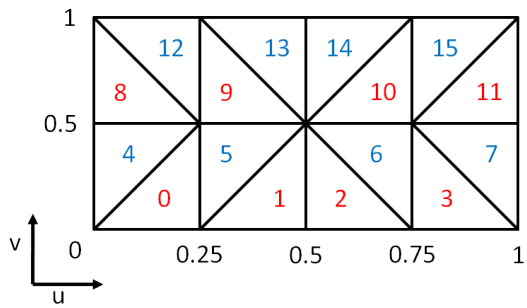


Figure 4.9: A quadrilateral patch is tessellated into $i \times j$ quadrilaterals with $i = 4$ and $j = 2$, each of which being subdivided into 2 triangles. We index the triangles depending on their barycentric tessellation coordinates u, v following the algorithm 2. Red indices correspond to triangle being the lower part of the quadrilateral, while blue indices are upper parts.

4.5 Terminal evaluator

The terminal set provides information regarding the location and parameters for the geometry associated to the terminals symbols. However, the structure of graphics hardware does not allow a direct substitution of the terminal set by the corresponding detailed geometry. Instead, we embed the vertex attributes of the terminal shapes within GPU buffers and reuse the triangle indexing scheme.

Using this representation, each primitive of the terminal structure is tessellated into the number of triangles corresponding to the target detailed geometry for the terminal. After assigning a unique identifier to each triangle, the geometry shader extracts the corresponding terminal geometry from the geometry buffers (Figure 4.10). Note that while geometry buffers provide a simple way of representing the terminal geometry, other approaches such as texture-guided subdivision surfaces could be used instead.

The generated geometry is finally rendered and shaded within the same render pass, hence avoiding explicit storage of the fully detailed model.

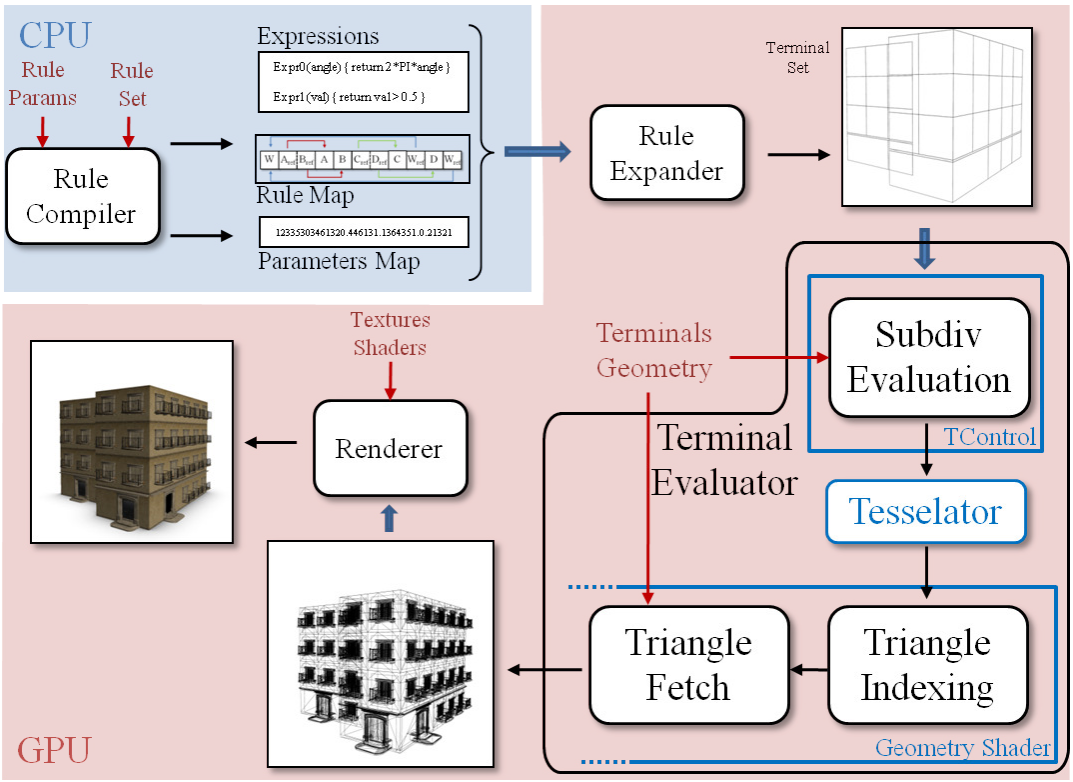


Figure 4.10: The detailed geometry of the terminal symbols is substituted to the terminal set using hardware tessellation and on the fly geometry evaluation, providing a fully detailed model ready for rendering.

4.5.1 Renderer

The output of our procedural generation pipeline is a simple set of textured triangle meshes which can be easily rendered using any state-of-the-art technique (Figure 4.11). Procedurally generated objects can therefore be combined to other scene components within a same render pass. Our method is then easily integrable into existing rendering engines. In particular, the existing deferred shading pipelines are left untouched. Finally, for the purpose of production rendering the output of the terminal evaluator can also be stored within GPU buffers and read back to files for later rendering.

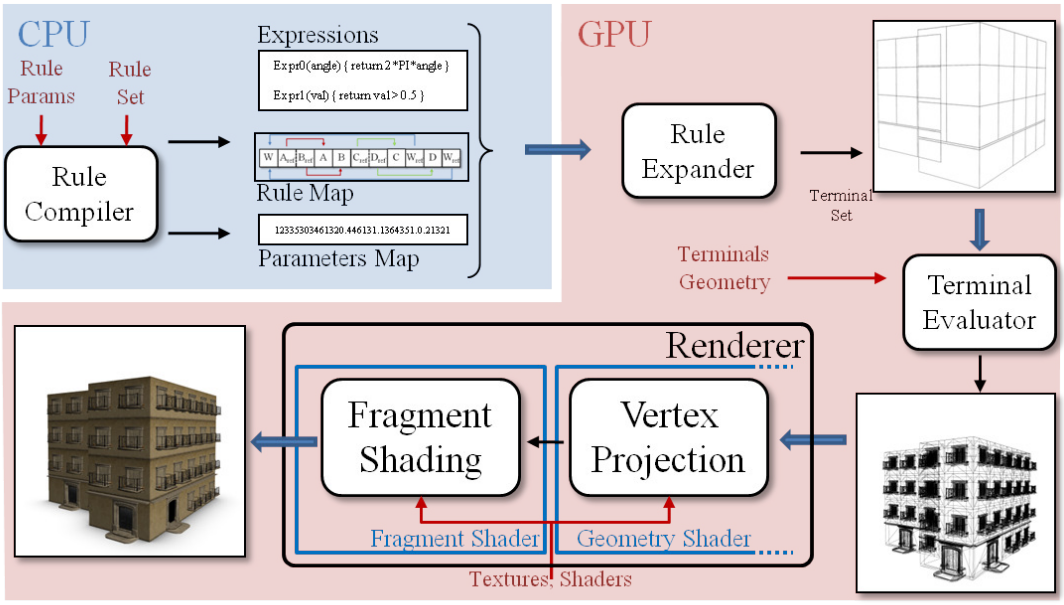


Figure 4.11: Rendering stage: our procedural generation pipeline generates a set of classical meshes ready for rendering using any existing shading technique.

4.6 Applications

Typical uses for procedural modeling include the generation of compelling buildings and plants. We assess our generic pipeline on architectural modeling and vegetation growth. Although these two kind of objects follow antagonistic modeling approach (*i.e.*, reduction *vs.* growth operations), we are able to efficiently generate them using our generic system. First, we describe the grammar-based generation of buildings and the segment-based strategy used. Then we see another approach for grammar-based modeling of plants.

4.6.1 Architecture

The CGA shape grammars [MWH+06] are a method of choice for procedural modeling of buildings. Starting from a footprint, the first steps of the grammar expansion generally consists in a small number of growth operations to generate the rough shape of the buildings. Then, potentially numerous reduction operations (*e.g.*, *split* and *component split*) divide each facade into a number of elements such as doors and windows. The generation parameters typically include the target number of floors, or the space between facade elements. A simple example of such grammar is provided below using a syntax following our segment-based formulation:

$$
\begin{array}{lcl}
\mathrm{W} & \rightsquigarrow & \mathrm{Extrude(buildingHeight)\{F, \oslash\}} \\
\mathrm{F} & \rightsquigarrow & \mathrm{Split(\text{"y"}, floorHeight, \sim)\{GF, FL_R\}} \\
\mathrm{GF} & \rightsquigarrow & \mathrm{Split(\text{"y"}, floorHeight\text{-}0.1, 0.1)\{G, T^L\}} \\
\mathrm{G} & \rightsquigarrow & \mathrm{Split(\text{"x"}, doorWidth, \sim)\{T^D, RW_R\}} \\
\mathrm{FL_R} & \rightsquigarrow & \mathrm{Repeat(\text{"y"}, floorHeight)\{RW_R\}} \\
\mathrm{RW_R} & \rightsquigarrow & \mathrm{Repeat(\text{"x"}, windowWidth)\{T^W\}} \\
\mathrm{T^L} & \rightsquigarrow & \mathrm{Shape(ledge)} \\
\mathrm{T^D} & \rightsquigarrow & \mathrm{Shape(door)} \\
\mathrm{T^W} & \rightsquigarrow & \mathrm{Shape(window)}
\end{array}
$$

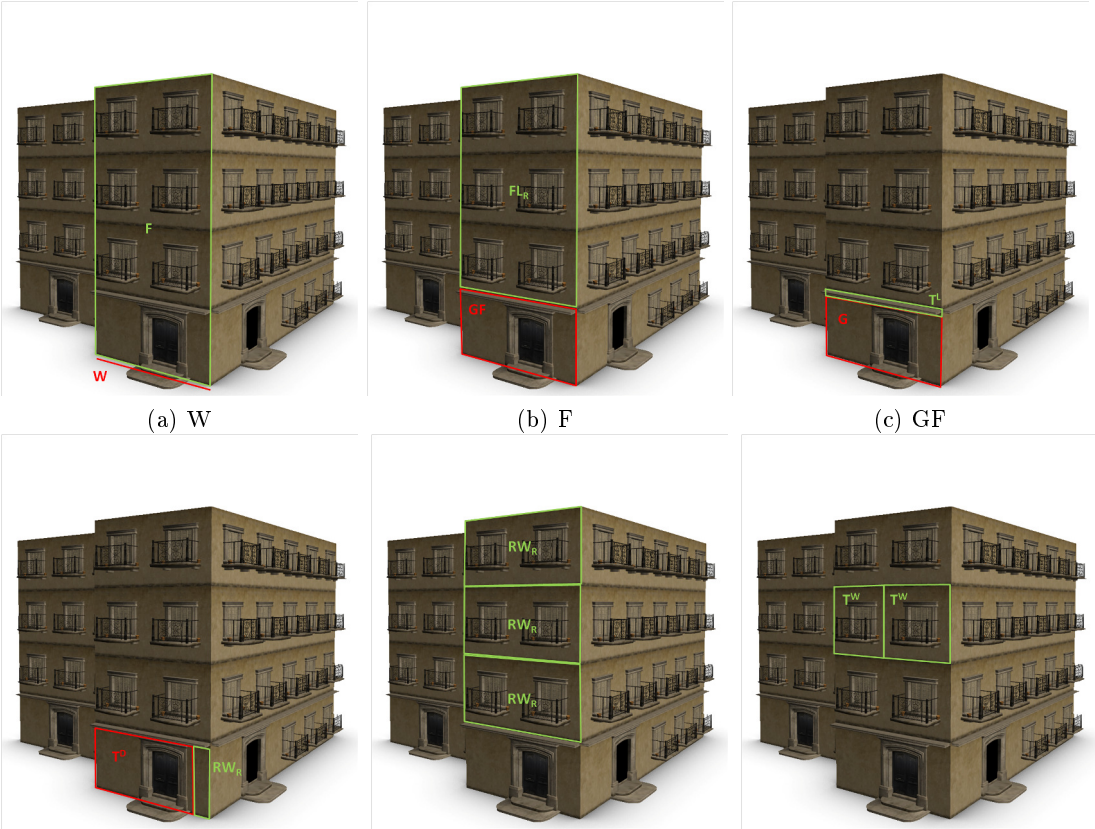
where F, GF, G and $\mathrm{FL_R}$ respectively represent the facade, ground floor including top ledge, ground floor elements and the other floors. $\mathrm{T^L}$, $\mathrm{T^D}$ and $\mathrm{T^W}$ are the terminal symbols of the grammar, linking to the actual geometry for the first floor ledge, the door and the windows. We illustrate in Figure 4.12 how this grammar is interpreted rule-by-rule to generate a building.

The split and repeat operations [MWH+06] are particularly useful in this context, and can be easily implemented in shader code. The geometry of the terminal symbols can be arbitrarily chosen and encoded into GPU buffers (Figure 4.13). Finally, assembling variety of terminal shapes yields a wide range of building appearances (Figure 4.14).

The input of the grammar is a set of generation parameters stored within the parameter map for fast GPU access. The footprint of the building is decomposed into a set of independent segments, bootstrapping our grammar expansion pipeline. While we feed our system with complex footprints, we automatically divide them into 1D atoms in a pre-processing step. Thus we benefit from the high number of threads interpreting the grammar in parallel.

4.6.2 Vegetation

The contrast between the generation of buildings and vegetation lies in the relative importances of growth and reduction operations. The very principle of plant evolution induces recursive, numerous growth operations, keeping reductions marginal. This expansion scheme seamlessly fits within the framework of GPU shape grammars, which support extrusion and implicit component split operations.



(a) W (b) F (c) GF

(d) G (e) FL_R (f) RW_R

Figure 4.12: Non terminal rules of the above grammar are explained on example. The axiom W is shown in a red line, while successors are blue and green quadrilerals, respectively for the first and second successors.

Figure 4.13: Many different terminal shapes are provided to increase the variety of object generated.

(a) (b) (c)

Figure 4.14: Our sample building grammar can be used to generate a variety of buildings following a same set of construction rules. Changing the geometry of the terminal symbols allows for further style variations. Rule expander takes 0.2ms per building, while terminal evaluation and rendering are performed in 2.6ms (a), 8.4ms (b) and 4.0ms (c). We observe these differences because terminal evaluation depends on the complexity of terminal shapes used.

The tree model[1] shown in Figure 4.15 is generated from the grammar on page 60 of [PLH+90], slightly modified to fit the framework of GPU shape grammars:

W	$\rightsquigarrow$	Extrude($trunkLen, twist_0$){T^{T}, Growth }
Growth	$\rightsquigarrow$	Cond($reclevel < n$){ BranchSplit, T^{B} }
BranchSplit	$\rightsquigarrow$	Branch(3){ Main, Sub1, Sub2}
Main	$\rightsquigarrow$	Rotate($q1$){ RotBranch }
RotBranch	$\rightsquigarrow$	Extrude($branchLen, twist_1$){T^{B}, LeafSet}
LeafSet	$\rightsquigarrow$	Branch(2){Leaf, Growth}
Leaf	$\rightsquigarrow$	Rotate(q_2){ T^{L} }
Sub1	$\rightsquigarrow$	Rotate(q_3){ B0 }
Sub2	$\rightsquigarrow$	Rotate(q_4){ B0 }
T^{T}	$\rightsquigarrow$	Shape(trunkGeom)
T^{B}	$\rightsquigarrow$	Shape(branchGeom)
T^{L}	$\rightsquigarrow$	CreateQuad(leafVertices)

The overall pattern of this grammar follows the principle of Figure 4.8, including recursive rules simulating the growth of smaller branches from the main ones. This recursion is represented within the rule graph and evaluated at run-time by our expression-instantiated interpreter, yielding procedurally-generated vegetation at arbitrary growth levels. In the case of vegetation, a single segment per tree is used as input for the grammar interpreter, leaving all the other parameters in the parameters map.

In both applications, architecture and vegetation, we take advantage of the paral-

[1] Terminal geometry and textures courtesy of `http://xfrog.com/`

lelism capabilities of graphics hardware thanks to the segment-based expansion. We push the concept farther by batching multiple inputs of a same grammar.

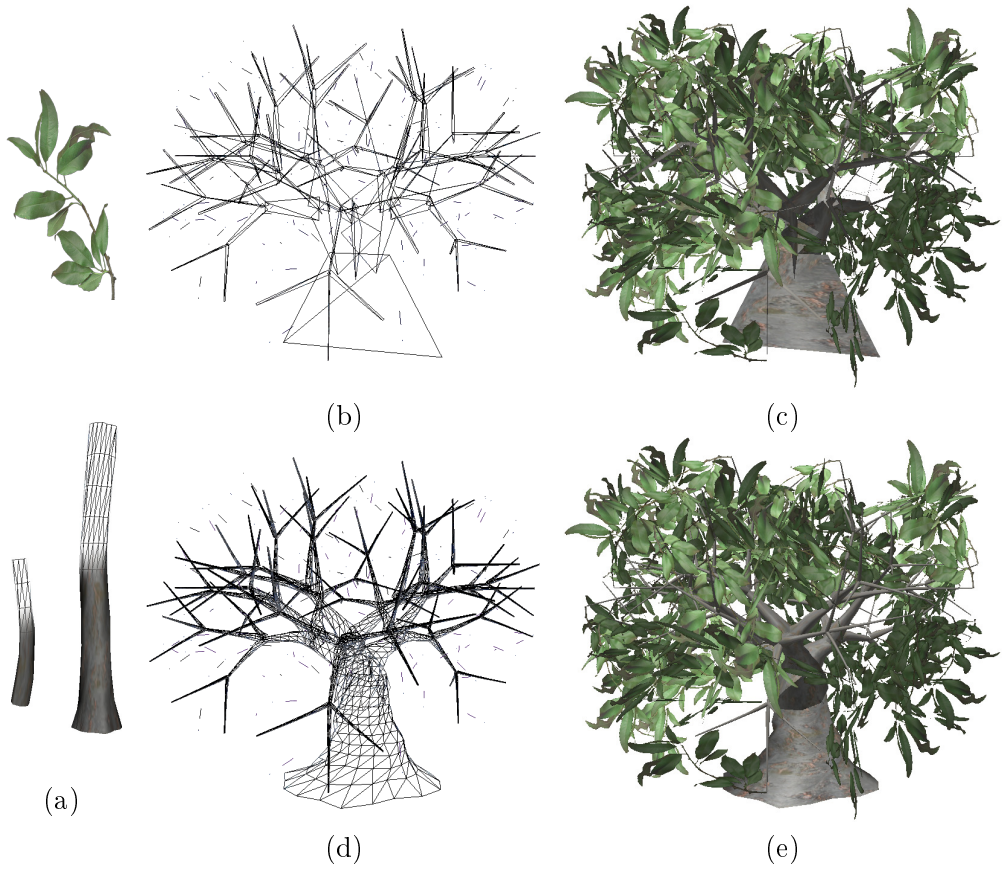


Figure 4.15: GPU shape grammars naturally apply to the generation of vegetation. Starting with an input segment and a small set of terminal shapes (a), the grammar is expanded on the GPU, yielding a set 240 terminal patches (b), on which textures can be mapped to generate a low level of detail (c). Those terminals are then replaced by their corresponding geometry, yielding a fully detailed model (d,e) generated from end to end in 40ms using GPU shape grammars. Once the terminal set is cached render speed reaches 530fps instead of 23.8fps with systematic regeneration.

4.6.3 Object batching & performance

Starting with a base geometry such as a footprint and a set of generation parameters our pipeline generates in parallel a terminal set describing the location and parameters of each terminal shape. However, sequentially repeating this process for each object prevents any parallel evaluation of the different objects, reducing the benefits of our method. As we design a generic expression-instantiated interpreter, the only difference between two instances of a same grammar lies in the input parameters. Therefore we

leverage the parallel architecture of graphics hardware by grouping the base geometries into a single vertex buffer (see Figure 4.16), and packing parameters into a *parameters map*. Thus, concurrent threads work on different input geometries fetch from the vertex buffer, but use the same grammar-specific interpreter and expression library. Finally we specify a unique id for each input geometry, so that the interpreter will fetch different parameters from the parameters map. The ith input geometry will fetch the ith values from the parameters map. Then, the terminal set for an entire set of objects with various parameters is generated using a single draw call. The terminal evaluation and rendering of the terminal shapes are also performed in parallel over the entire terminal set.

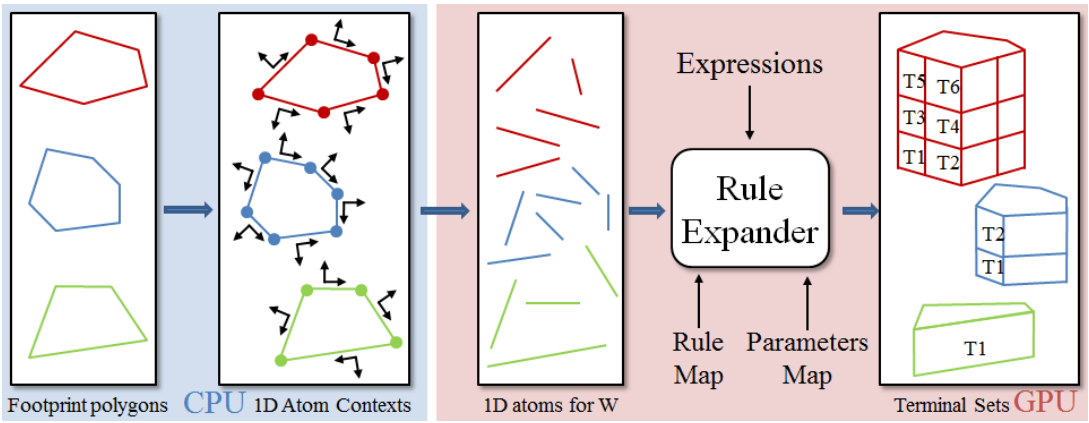


Figure 4.16: All the footprints are associated to a unique grammar, divided into independent 1D atoms and regrouped into a single buffer. As the parameters are also packed together in a buffer, the expansion is performed for each 1D atom in parallel in a single draw call.

Consequently, multiple objects can be generated and rendered at the same cost as a single object until all computing cores are filled. As an example, we measured the generation time for various building following the same grammar and batched together. Figure 4.17 reports a chart with the generation time depending on the number of input segment. From this chart we observe a repetitive pattern composed of a short period where the generation time keeps increasing linearly (green periods: A and C), followed by a larger period where the generation time is stable (red periods: B and D). This pattern is repeated over the input segment count (A/B, then C/D, etc). This can be interpreted according to the parallelism provided by the graphics hardware. At the beginning, when only few segments are given in input to the pipeline, the GPU prepares a first computing grid and then processes it. As the generation are performed in parallel, computation time should be stable from 1 to 20 input segments. However, the time spent to the grid preparation depending on the number of input, the final generation time (*i.e.*, preparation and computation) grows linearly on the number of input. Then, a stable period is reached from 20 to 100 input segments (period B). Available computing grids are prepared in parallel from the first one, involving a stable the generation time.

Finally, another increasing section is observed (period C). It suggests the GPU may processed up to 100 segments in parallel for this case. When the GPU is filled with more than 100 inputs, it prepares other computing grids that will be processed in another round of computation, after processed the previous ones. We thus observe the beginning a second round of parallel computation.

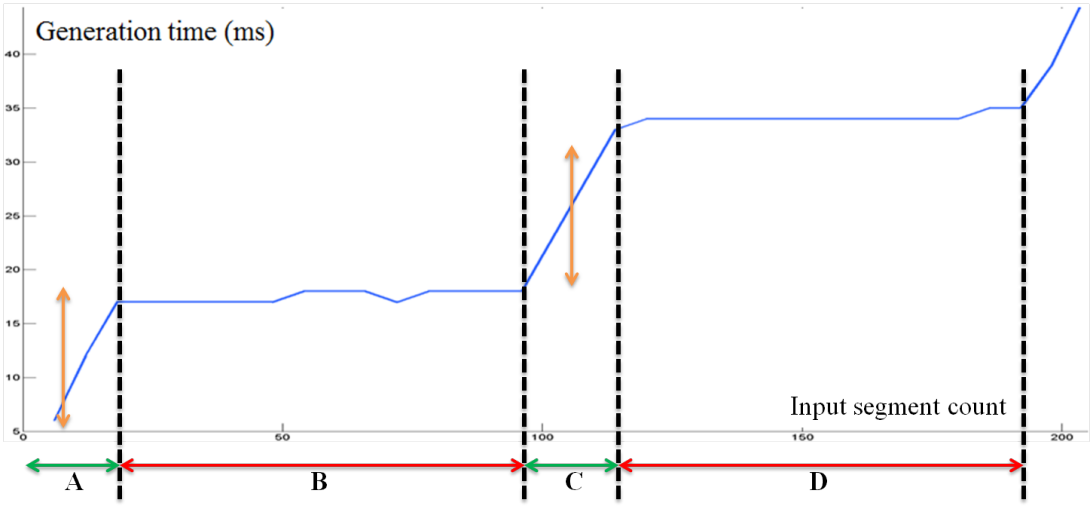


Figure 4.17: Sublinear complexities are observed when the GPU shape grammars is filled with many input segments. Around 100 input segments are processed in parallel at each computing round. The orange arrows show the preparation time of one computing grid. For a given round of computation (*e.g.*, A+B, C+D sections), the first linear slope is due to the first grid preparation. Then, other grids preparation are performed in parallel to the first one.

Although our approach, based on a highly parallelized formulation for grammar expansion, allows for efficient parallel generation, especially when we batch objects interpreting a same grammar together, it also has a counterpart. As we decompose input footprints into independent 1D atoms, the footprint context (intra-object) is entirely lost. Each 1D atom performs the grammar interpretation autonomously, without any information about the initial footprint. Moreover, information about neighboring footprints (inter-objects) is also unavailable in our system. The lack of inter and intra-objects structure information prevents from internal and external context sensitive operations such as snapping and occlusion operations. These functionalities adapt the generation of procedural models to their environments [MWH+06]. Typical example are the alignment of the window levels (snapping) and the avoidance of openings in occluded building facades. While those operations are costly in CPU-based generation schemes, our segment-based formulation for parallel grammar expansion prevents such techniques to be applied within a single render pass.

4.7 Conclusion

We introduced GPU shape grammars, a generic solution for real-time generation, tuning and rendering of procedural models. Based on an expression-instantiated rule interpreter coupled to parallel segment-based grammar expansion, our approach generates geometry on the fly within graphics hardware. The generated models are then directly streamed across the graphics pipeline, avoiding the storage of the fully detailed models.

We demonstrated the application of our method for real-time generation of both buildings and vegetation. GPU shape grammars generate and render relevant geometry during navigation, allowing for simultaneous navigation and parameters tuning. We also use our pipeline for real-time generation of terabyte-sized urban environments, the problem of massive sceneries being addressed in Chapter 7.

The fast rendering capabilities of our method find a particular interest in interactive applications, in which memory consumption and computational efficiency are critical. Also, the ability for interactive tuning and rendering of arbitrarily massive models makes the GPU shape grammars a highly valuable tools within the visual effect and game industries.

However, to benefit from the high number of threads running concurrently, we automatically decompose input complex footprints into independent 1D atoms. This decomposition leads to the loss of internal context information, and makes impossible the generation of models based on internal structures, for instance when addressing the problem of the roof generation. As 1D atoms are expanded independently, they cannot converge consistently without any additional information, preventing consistent roof generation over the initial footprint. In the next chapter, we consider the processing of such internal context operations, with respect to the independent 1D atoms and propose an efficient solution to preserve parallel processing capabilities of our pipeline.

Chapter 5

Internal context parallelization
and application to roofs modeling

(a) Hipped with halfway dormer (b) Haussmann-type (c) Normandy

Figure 5.1: We extend the GPU shape grammars pipeline to reconstruct structured data. Our solution allows for efficient modeling of complex roof structures starting from independent 1D atoms. Note that gutters, chimneys, ridge tiles and overhangs are also generated by the system.

Application of our procedural pipeline introduced in Chapter 4 to model structured data. We feed the independent 1D atoms with local context information to generate internally consistent models at run-time. All along this chapter, we take the example of a highly detailed roofs. Starting from a consistent roof structure such as a straight skeleton computed from the building footprints, we decompose this information into local roof parameters per input segments compliant with GPU shape grammars pipeline. We also introduce *Join* and *Project* rules for a consistent description of roofs using grammars, bringing the massive parallelism of GPU shape grammars to the benefit of generation of consistent structures. This work has be published in the GPU roof grammars paper [BMG13].

5.1 Introduction

The procedural pipeline introduced in Chapter 4 intensively uses the parallelism capabilities of GPUs. Our approach relies on a reformulation of the classic expansion scheme to a segment-based one. By decomposing input footprints into 1D atoms, many 1D atoms are processed in parallel thanks to our expansion scheme. While enforcing parallelism, such decomposition leads to the loss of the internal context of the input data. As an example, once a footprint building is exploded, each 1D atom is entirely independent from its neighbors and the internal structure of the footprint remains unavailable during the expansion process. Internal context models such as roof structures can not be generated using GPU shape grammars pipeline as-it-is and require ad-hoc solutions.

Roof construction requires a global processing of the building footprints to extract the ridges and gables. Among the solutions designed for structure extraction from a polygon, Aicholzer *et al.* introduced straight skeleton algorithm [AAAG96]. This algorithm computes the skeleton of the polygon by sweeping the edges according to the bisector angles. This technique has been applied for urban modeling purposes in various methods for generation of complex structures such as mansard and hipped roofs [LD03, MWH+06, KW11]. However, due to the strongly sequential nature of this algorithm, most GPU methods for procedural architecture compute roofs on the CPU and transfer the geometry to the GPU for rendering [HWA+10, MBG+12]. To address roof generation, we push further the segment-based expansion from the facades to the roofs, where each 1D atom generates both its facade and its roof. More generally, we generate internally consistent objects on the GPU by feeding 1D atoms with local information to guide the construction. This per 1D atom local context information allows to simulate structured input data.

Our contributions bring parallelism to grammar-based generation of consistent models (Figure 5.2). First, a CPU component converts consistent internal contexts computed from building footprints into local contexts information consistent with the 1D atoms paradigm (see Chapter 4). In the case of roof generation, this information typically include the target roof height and slopes. We also enrich GPU shape grammars with *Join* and *Project* rules to guide creation of consistent model using the information distributed among the 1D atoms. Following the GPU shape grammars pipeline, we evaluate the new rules at interpretation stage to benefit from the parallel processing capabilities of the graphics hardware.

5.2 Principle

The idea of this work is to let 1D atoms handle their own section of a internal consistent model. In order to reconstruct a consistent internal information from independent 1D atoms, we guide unstructured 1D atoms with a local context extracted from the internal context. To do this, we integrate four new steps within the GPU shape grammars pipeline presented in Chapter 4. We first determine an internal consistent structure on the CPU from an input geometry in (see Section 5.2.1, Figure 5.2a). We then decompose

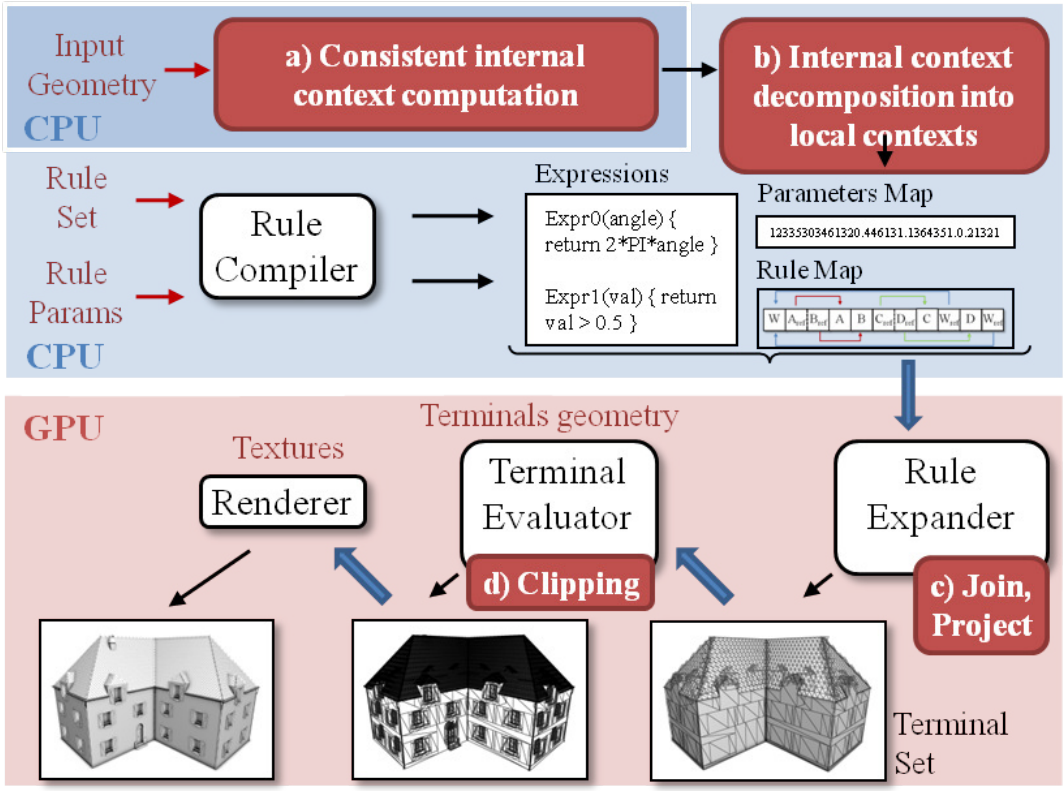


Figure 5.2: Overview of our method used for generation of internally consistent models. New steps are indicated in red. a) We compute a consistent internal context from the input geometry. b) Conversion of internal context into 1D atoms local context. c) We introduce *Join* and *Project* rules to generate consistent structure from independent 1D atoms in parallel. d) Finally terminal evaluation stage performs the clipping of shapes issued by the *Join* rule.

this information into local context for 1D atoms (Section 5.2.2, Figure 5.2b). Multiple 1D atoms may thus generate in parallel their section of the initial internal model. However, rules provided in GPU shape grammars are not sufficient enough to model consistent structure from independent 1D atoms. We thus add *Join* (Section 5.2.3) and *Project* (Section 5.2.5) rules to the rule expansion stage (see Figure 5.2c). Finally, a clipping operation required by the *Join* rule is performed during terminal evaluation (Section 5.2.4, Figure 5.2d).

We apply our method on the roof generation from input footprints. In this case, we start from the input footprint to compute an internal consistent context which is the roof skeleton. Then, we feed each 1D atom with their local context extracted from the roof skeleton. They are thus able to generate in parallel their roof section using the local context that finally result in a consistent roof model. We illustrate our approach

with the roof generation case all along the remaining of this chapter.

5.2.1 Internal consistent context computation

The first step to build consistent roof from unstructured 1D atoms is to compute the internal consistent context from the input geometry. We consider this information to be preprocessed by any offline CPU-based algorithm (Figure 5.2, top left box). For instance, we may fill the internal context with the straight skeleton algorithm implementation from CGAL[1] or user-defined inputs. Figure 5.3 shows different results given by the straight skeleton algorithm computed on closed polygons. We use this information as destination locations for 1D atoms. We thus extract the internal context into per 1D atom local context.

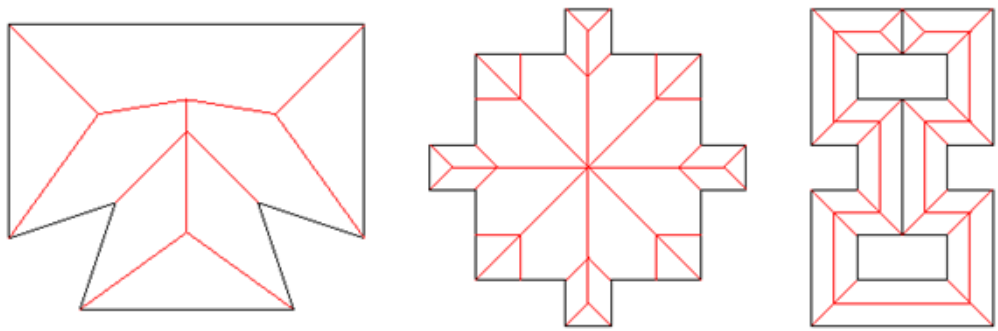
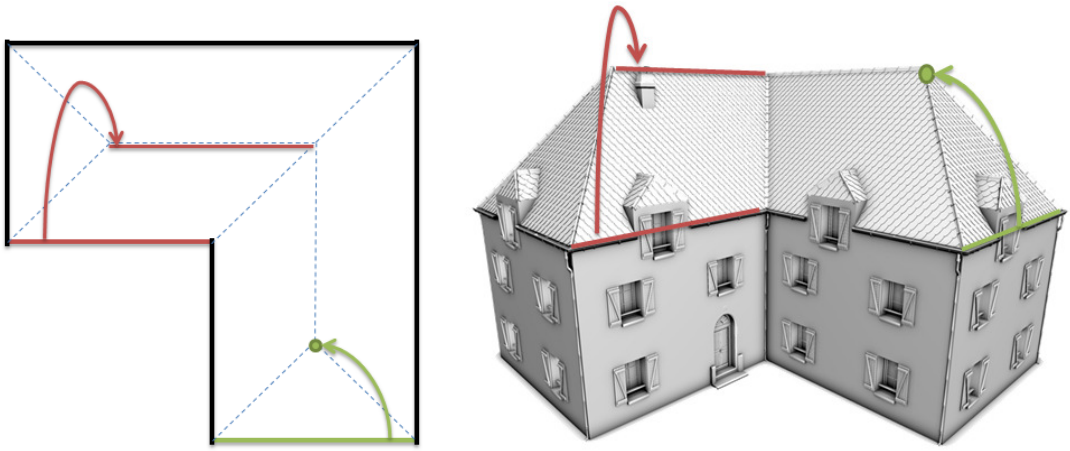


Figure 5.3: Straight skeleton (red) computed on different polygon footprints (black). Image courtesy of CGAL 1.

5.2.2 Internal context decomposition

Previous internal consistent context is computed on the entire footprint polygon, given the overall shape of the roof skeleton that will be used for the consistent roof generation. We then need to decompose the output of the CPU-based internal context generator into multiple local contexts. To this end we associate one section of the internal context to each segment of the input footprint as its local context, and store this information into the parameters of the 1D atoms (Figure 5.2). For the roof generation, we associate one roof section destination for each 1D atoms so that all roof sections are generated consistently according to the internal context. We consider two cases of 1D atom roof section: the segment either reaches another segment, or a vertex (Figure 5.4). In terms of roof architecture, it corresponds to ridge (segment) and gable (vertex) cases. We excluded cases where the target is composed of two or more segments to avoid T-vertices. This point will be discuss later in Section 5.5. From this observation both cases can be reduced to a segment to segment operation, where a gable is obtained by duplicating the destination vertex.

[1] http://www.cgal.org



(a) Top plane view (b) 3D view

Figure 5.4: Starting from on closed building footprint (solid lines), an off-line algorithm computes its roof skeleton (dash lines). Each 1D atom is then associated with a destination on the skeleton according to two cases: ridge (red) or gable (green). Left: association cases on the plane. Right: final 3D roof construction.

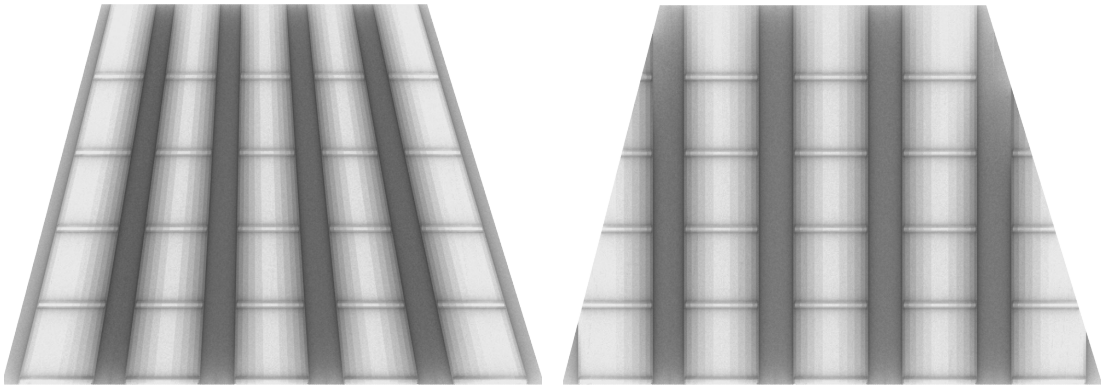
5.2.3 Join rule

Once we have specified the local context for each 1D atoms, we may reconstruct the internal consistent context at the rule expansion stage using grammar rules. The only available rule in Chapter 4 to fill a non rectangular quadrilateral with a set of shapes is the *Extrude* rule. However, as soon as the extrusion does not generate a rectangular quadrilateral, the geometry mapped is distorted to fit the quadrilateral (Figure 5.5a). In some applications such as synthesizing a branch geometry onto a scaled extruded quadrilateral, we actually desire the geometry to be distorted in order to fit the space. Nevertheless, there are also some cases such as the roof tile covering where we do not want to introduce such distortions. Instead we would like to simply clipped the geometry by the edges (see Figure 5.5b). Thus, to avoid distortions we introduce the *Join* rule:

Pred→Join(float heightExtrusion, vec2 v1, vec2 v2)
{quadClipped, quadSupport, segTop}

where *Pred* is the rule predecessor, *quadClipped* is the rule successor applied onto the clipped extruded quadrilateral (Figure 5.6b), *quadSupport* is the rule successor applied onto the joined face (Figure 5.6a) and *segTop* is the rule successor applied onto the extruded segment (Figure 5.6c). *heightExtrusion* is the height of the extrusion, while the target 2D segment is represented by *v1* and *v2*.

Therefore, the *Join* rule acts as a clipped extrusion rule guided by a 2D segment destination and a height. Moreover, unlike the scaled *Extrude* rule, *Join* does not change the surface parametrization but prepares some clipping planes (see Figures 5.6d-f) that are used at terminal evaluation stage (Figure 5.2).



(a) Scaled *Extrude* rule (b) *Join* rule

Figure 5.5: Differences on surface parametrization.

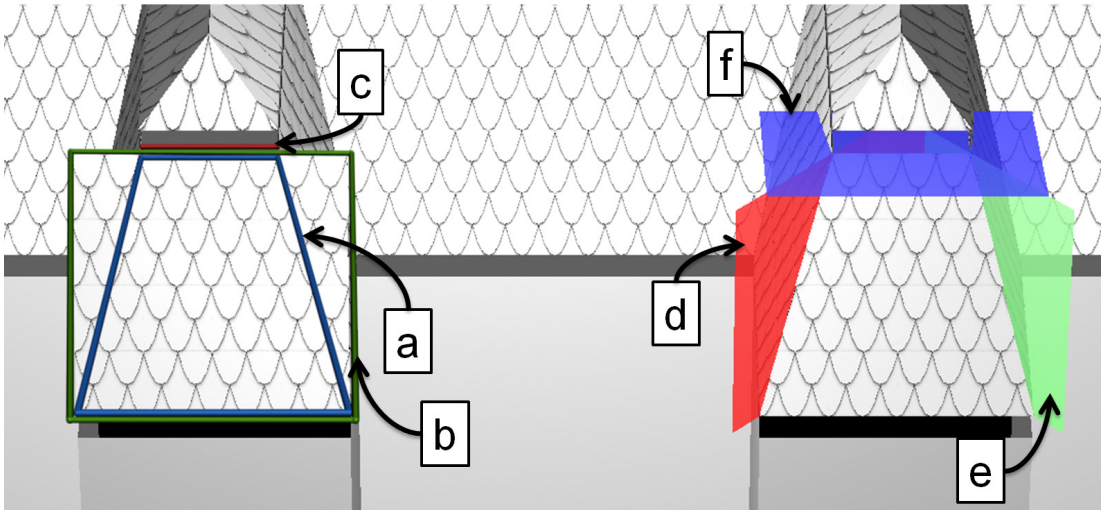
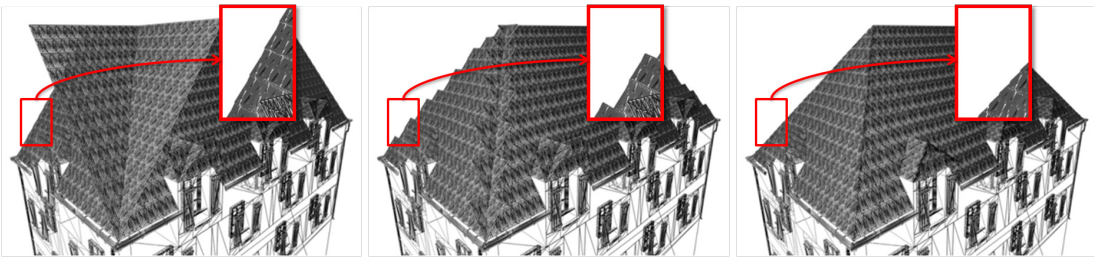


Figure 5.6: The *Join* rule has three successors : *a)* a joined face, *b)* an extruded rectangular quadrilateral including the joined face, and *c)* the top segment defined by *v1* and *v2* at *heightExtrusion* added to starting height. Clipping planes are defined around the joined face and applied on the extruded quadrilateral. The left *d)*, right *e)* and top *f)* clipping planes are oriented according to the bissector angles.

5.2.4 Clipping

Clipping planes are automatically generated after a *Join* rule and applied to the extruded rectangular quadrilateral. These clipping planes are oriented according to the bisector angles (Figures 5.6d-f), so that adjacent terminal shapes are consistently assembled. Patch clipping is partially performed at the rule expansion stage for the

terminal shapes generated from a *Join* rule (Figure 5.7a to b). Fully clipped terminal shapes are simply not generated. Triangle clipping is achieved at terminal evaluation stage by applying an optimized clipping algorithm [McG11] within the geometry shader for all terminal shapes being crossed by a cutting edge (Figure 5.7b to c). This algorithm takes advantage of the SIMD capabilities of the graphics hardware to efficiently eliminate clipped triangles.



(a) No clipping (b) Patch clipping (c) Triangle clipping

Figure 5.7: Clipping is performed in two passes. First, non visible terminal shapes are clipped at rule expanding stage. Then, terminal shapes concerned by cutting edges are triangle-clipped.

5.2.5 Project rule

During rule expansion one may want to project the current quadrilateral onto a chosen plane, for instance to create a balcony window as part of a roof section (Figure 5.8). While splitting the roof section, the roof part that will become the balcony can be separated. As this roof part is originally aligned with the roof surface, it can projected onto the plane defined by input axes, typically the local tangent of the quadrilateral and the cross product between this local tangent and the normal of the input footprint. The projected quadrilateral can then become the base of a balcony. We thus introduce the *Project* rule:

$$\text{Pred} \rightarrow \text{Project(vec3 axis1, vec3 axis2) \{quadProjected\}}$$

where *Pred* is the rule predecessor and *quadProjected* is the rule successor applied onto the projected quad. The plane is defined by the vectors $axis1$ and $axis2$.

The height of projection is made available through the current-scope built-in accessor *projectionHeight()*. The tangent, binormal and normal of the input building are set at the beginning of the grammar evaluation, while the local base is updated throughout the rule expansion. This information is also available for the user through accessors such as *localTangent()* or *globalNormal()*. Users may thus easily set the desired projection.

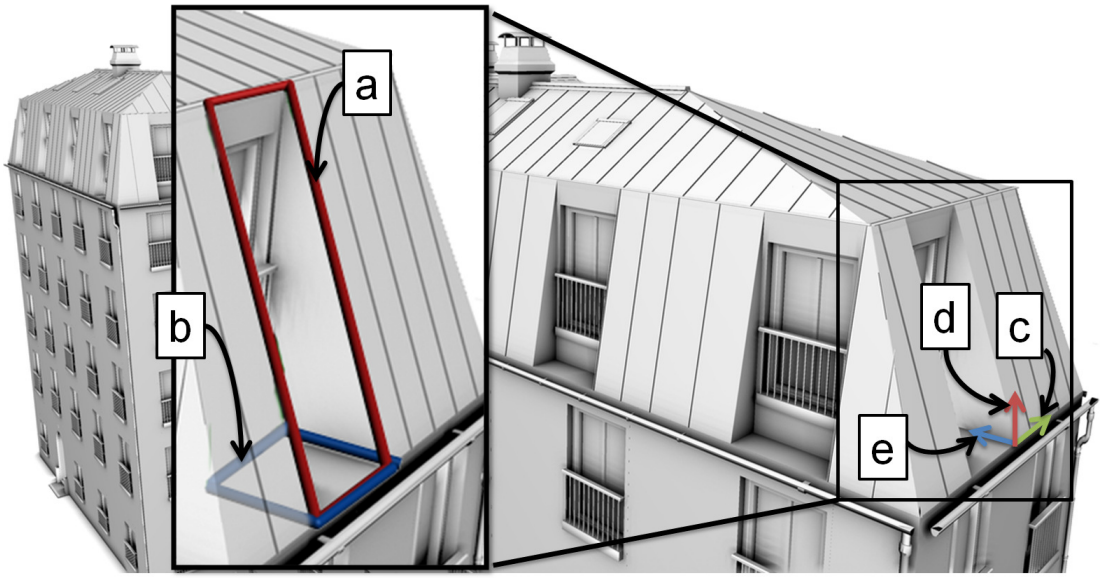


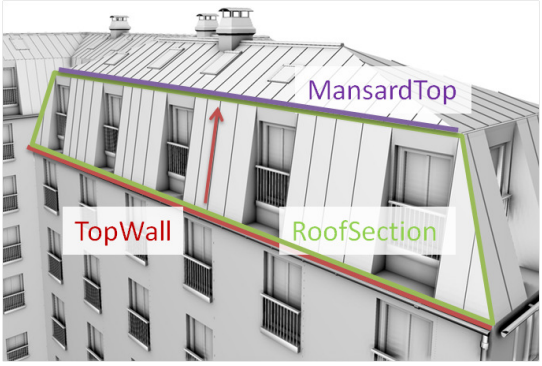
Figure 5.8: Illustration of the *Project* rule. The blue quadrilateral *b)* is the projection of the red quadrilateral *a)*. The projection axes are the local tangent *c)* and the cross product *e)* between the local tangent and the global normal *d)*.

5.3 Case study

Our method allows to procedurally generate compelling roofs of different architecture styles. For instance the following rule sequence describes a roof with a balcony window, using *Join* and *Project* rules coupled with local context information. This sequence corresponds to the first roof section of the roof part detailed in Figure 5.9.

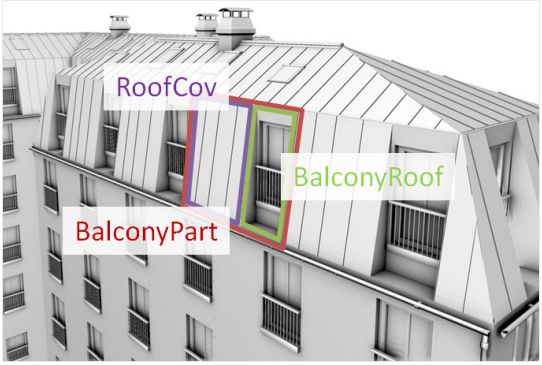
TopWall	$\rightsquigarrow$	Join(3, roofPos1, roofPos2) { RoofSection, NULL, MansardTop }
RoofSection	$\rightsquigarrow$	Repeat(X, 3) { BalconyPart }
BalconyPart	$\rightsquigarrow$	Split(X, 2, 1, $\sim$) { RoofCov, BalconyRoof, RoofCov }
BalconyRoof	$\rightsquigarrow$	Project(localTangent(), cross(localTangent(), globalNormal())) { BalconyFlat }
BalconyFlat	$\rightsquigarrow$	Explode() { FloorShape, NULL, RightBalc, BackBalc, LeftBalc }
RightBalc	$\rightsquigarrow$	Join(projectHeight(), currentPos1(), currentPos1()) { WallBalcShape, NULL, NULL }
LeftBalc	$\rightsquigarrow$	Join(projectHeight(), currentPos0(), currentPos0()) { WallBalcShape, NULL, NULL }
BackBalc	$\rightsquigarrow$	Join(projectHeight(), currentPos0(), currentPos1()) { NULL, WindowShape, NULL }
RoofCov	$\rightsquigarrow$	Repeat(XY, sizeTiles) {TilesShape}

The hole created by the *Project* rule *BalconyPart* is entirely filled by reconstructing the missing parts using the *Join* rules *LeftBalc*, *RightBalc* and *BackBalc*. Note also the *Explode* rule that allows to call distinct successors on the current quadrilateral, and each of its four segments.

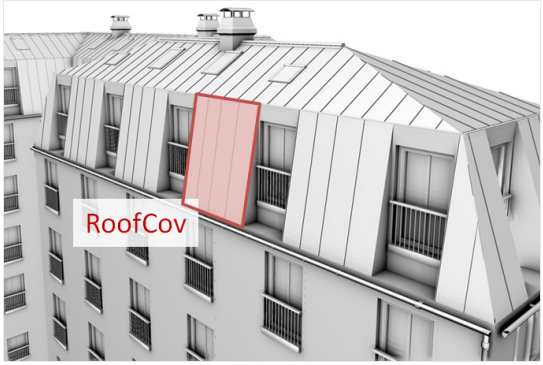


(a) Top Wall

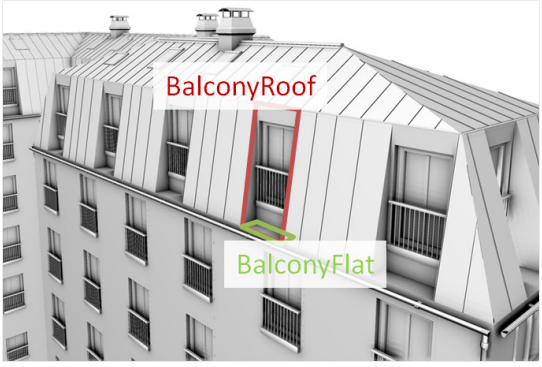
(b) RoofSection



(c) BalconyPart

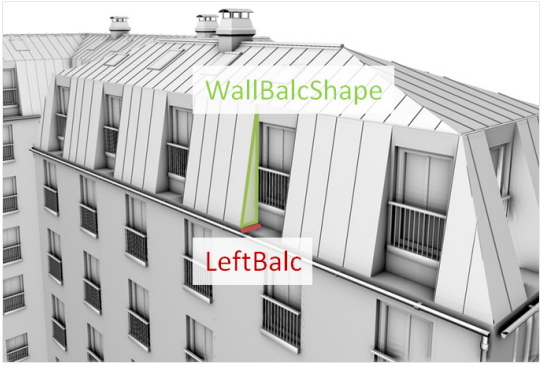


(d) RoofCov

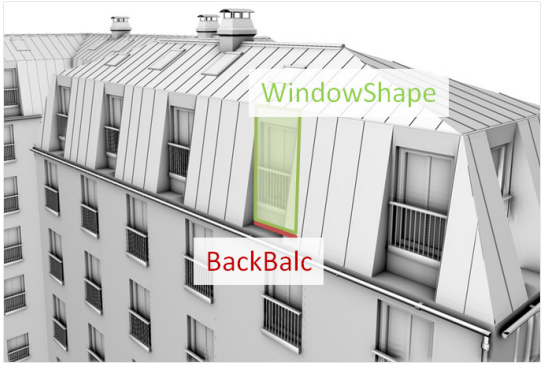


(e) BalconyRoof

(f) BalconyFlat



(g) LeftBalc



(h) BackBalc

Figure 5.9: The roof grammar presented in this chapter is described rule by rule. Predecessor is indicated in red, while successors are either in green, blue or purple. *RightBalc* rule is similar to the *LeftBalc*. Terminal shapes are applied on color filled quadrilaterals.

While this example illustrates how to create this style of roof, the *Join* rule is also useful for modeling other consistent architectural objects such as gutters. Particularly extremities may be seamlessly joined according to the angles computed from provided per 1D atoms local context information. In this case, the same local context information is used to generate both roof sections and gutters. While for creating the roof sections we *join* toward the local context (*i.e.*, the roof skeleton), we simply use the inverse direction to generate gutters. Gutters are thus created on the opposite side and clipping planes provide seamless geometry mapping (see Figure 5.10).

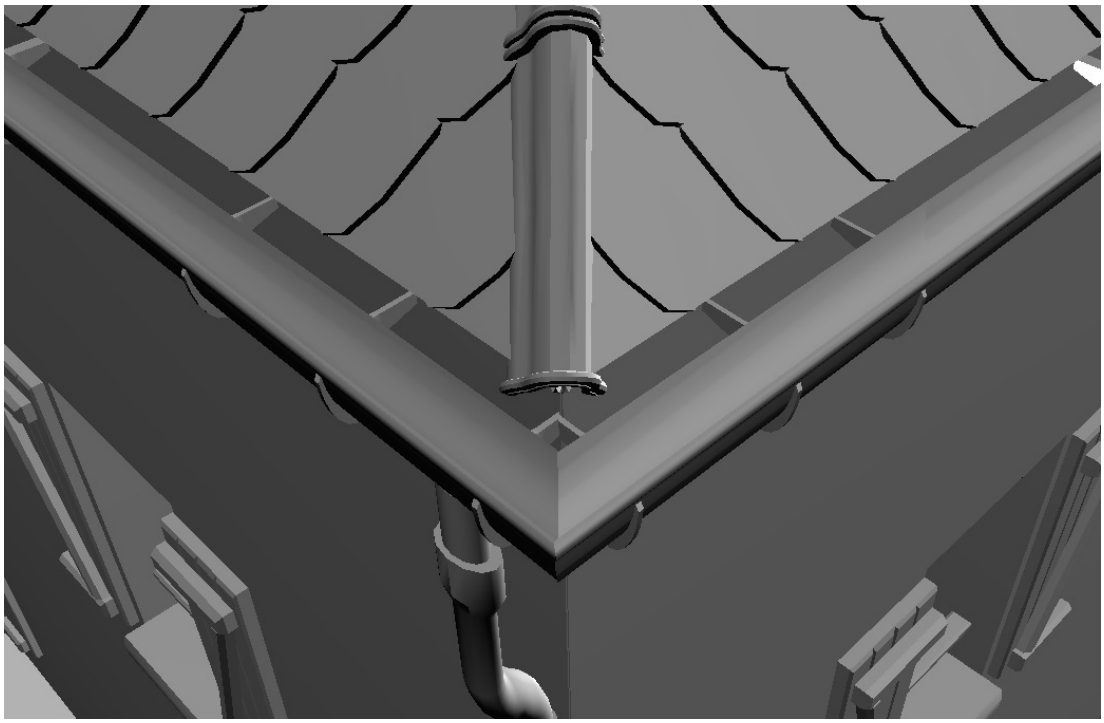


Figure 5.10: Gutters are consistent models which use the same local context information as for roof construction. We benefit from seamless geometry mapping between adjacent gutters using the *Join* and the automatically generated clipping planes.

5.4 Applications & results

We modeled various complex roofs structures featuring highly detailed geometries using *Join* and *Project* rules. Different architectural roof styles may be modeled using our method such as mansard, gable, hipped, overhangs and others. Roofs are covered with different geometric shapes such as slates, flat or round tiles provided by artists, using *Repeat* rules over the roof sections. We also modeled geometric roof elements that are

essential to create compelling roofs (Figure 5.11). For instance, chimneys, dormer or flat windows, balconies and ridge tiles are also modeled in a few rules.

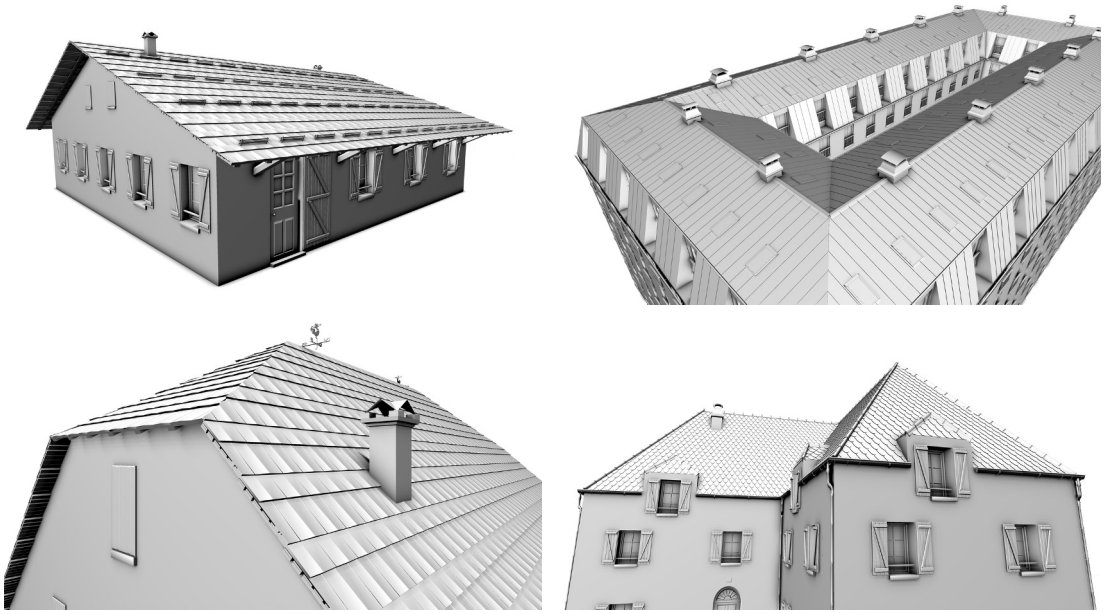


Figure 5.11: Simple use of the grammar rules allows us to create dormer windows, flat windows, roof balconies, chimneys, ridge tiles, gutter, beams, and more.

As for models of Chapter 4, roof parameters can be tuned interactively. We measured timings on object generation (rule expanding stage), and on terminal evaluation, clipping and rendering using a NVIDIA GeForce GTX 480 GPU. We observed that buildings (facades and roofs) are expanded in sub-linear complexities, confirming Chapter 4 results. As we expand objects in parallel, we may generate many objects at a reduced cost. For instance expansion of 1 to 20 Normandy-type houses (80 segments) requires a constant time of 11.5 ms. The expansion of 21 to 40 houses (160 segments) then takes 22.3 ms. Terminal evaluation, clipping and rendering scale linearly, taking around 1.02 ms per input segment.

5.5 Discussions & future works

We experimented two strategies for geometric roof covering: the repetition of many terminals yielding a single tile each and the use of fewer terminals leading to batches of tiles (2×2 tiles). Rule expanding is thus faster in the second case. For instance, the halfway dormer roof of Figure 5.1 is expanded 1.92× faster using batches of tiles. Also, the terminal evaluation part runs 1.1× faster than using unit tiles. Hardware tessellator seems to prefer few denser meshes than many sparse meshes.

In counterpart, misalignment can occur in some specific cases (Figure 5.13). While

the *Repeat* rule creates an even repartition across adjacent children, the *Split* rule may introduce irregularities when the splitting and repetition axes are not identical (Figure 5.12a-b). The children are either scaled to fit the space, or translated to be aligned with the split (Figure 5.12c). To limit these distortions (see Figure 5.13) we can automatically adapt the number of repetitions to the nearest integer part (Figure 5.12d). However the ideal solution would be to globally distribute the children of repeat rules and clip geometric parts lying outside the split axis (Figure 5.12e). Moreover, we could use the clipping stage introduced at terminal evaluation to perform this operation.

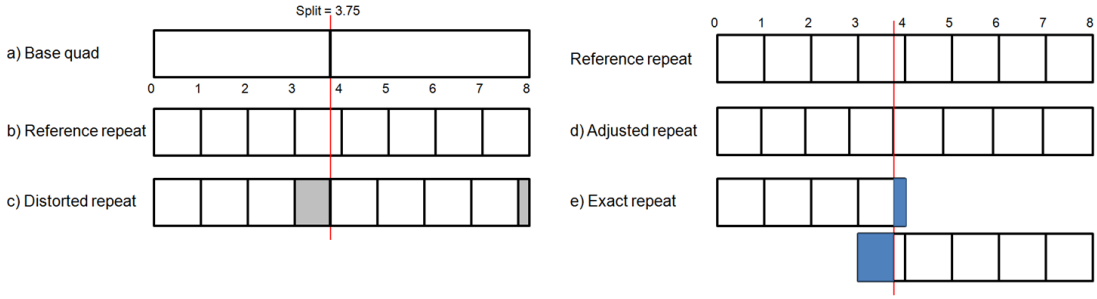
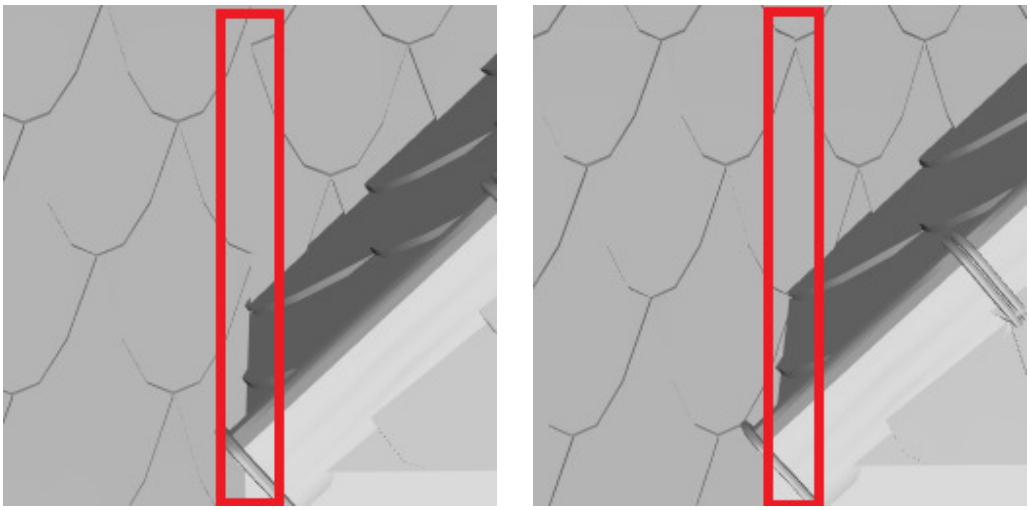


Figure 5.12: Misalignment occurs when repeat and split axes are not identical. A base quadrilateral of size 8cm is split at 3.75cm, then a part is repeated every 1cm *a)*. The split axis is indicated in red. The reference case shows repeat parts seamlessly aligned *b)*. In GPU shape grammars last element of each split part is scaled to fit the remaining space exhibiting artifacts (*c)* in grey), and first element of the second split part is translated to be aligned with the split axis. Repetition sizes may be modulated to limit the distortions (*d)*). To respect reference pattern, we would need a global distribution of the repeat parts and clip the ones lying outside the split axes (*e)* in blue).



(a) Misalignement (b) Reference

Figure 5.13: Tiles being on the border of the split axis are not well aligned *a)* compared to the reference case *b)*.

Finally, the straight skeleton algorithm may lead to incoherent, unrealistic roof structures. In some cases, one input segment can be associated to multiple destination segments, currently not supported by GPU roof grammars. Nonetheless our approach can be associated to any other user-defined roof structure extractor. Future work could therefore consider new skeleton generation algorithms based on architectural rules and constraints. Such skeletons could be directly used as input of our pipeline to model highly detailed roofs.

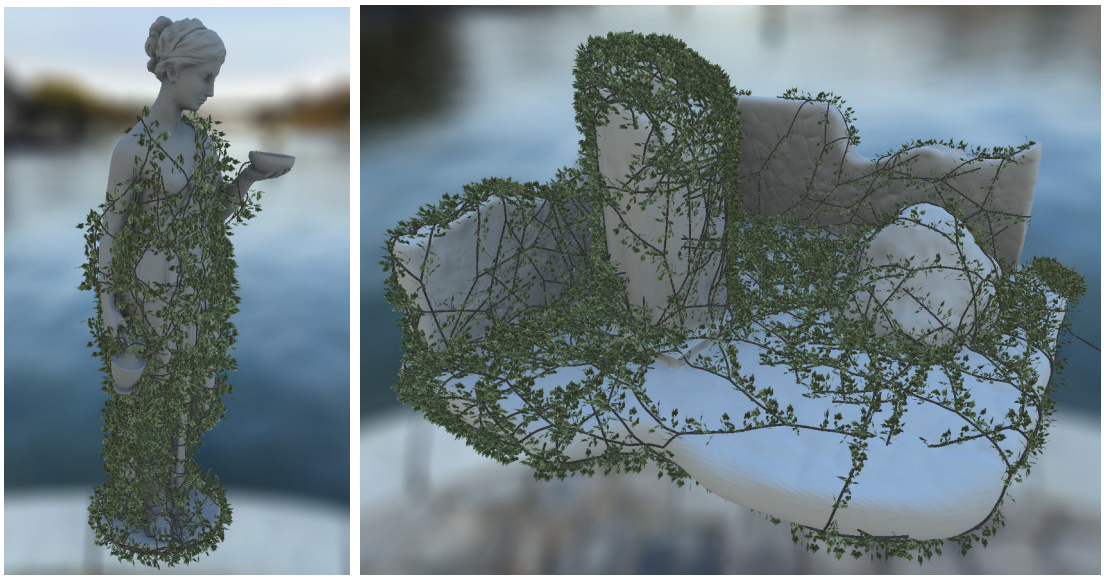
5.6 Conclusion

This chapter introduced a solution to create internal consistent models from independent local 1D atoms. In Chapter 4, the input footprints were decomposed into independent 1D atoms, preventing the generation of internal consistent models. Starting from an initial footprint, its internal consistent context is computed. Then, for each 1D atom, the local context information is extracted from the internal context information. Thus the 1D atoms are able to reconstruct their section of the internal context. We assessed our approach on architectural modeling where we compute a straight skeleton algorithm corresponding to the internal consistent context of the input footprint. Then we decompose this internal information into per 1D atom local context. Each atom is thus able to generate both its own facade and roof section using proposed *Join* and *Project* rules. The clipping stage introduced at terminal evaluation allows for real-time geometric clipping, providing seamless geometric mapping. Using the same approach, consistent context objects such as gutters may be generated all around the footprint with seamless clipping. We illustrate our method by creating various styles of compelling roofs. As our solution is fully integrated within the GPU shape grammars pipeline, we benefit from the parallelism capabilities of the GPU. Results show sub-linear complexities similar to the one noticed in Chapter 4, confirming our internal context parallellization approach.

While this approach of per 1D atom internal context allows for generating structured data, it is limited to simulate intra-objects context. For instance, we cannot use information of an underlying surface, either geometric or texturing, during rule expansion. In the next chapter, the challenging problem of external contexts guiding the grammar evaluation is addressed.

Chapter 6

External context-sensitive grammars
and application to growth on generic shapes



(a) Hebe model (b) Ruins

Figure 6.1: Surface contexts are used to expand grammars over an underlying surface. Texture contexts help to constrain grammar rules.

Controlling grammar rules from external contexts is addressed in this chapter. The GPU shape grammars pipeline is extended to handle texture and surface contexts. This solution allows to constrain any grammar rule with texturing or geometric information. In addition, our system provides on-the-fly painting for artists with interactive feedback.

6.1 Introduction

GPU shape grammars pipeline introduced in Chapter 4 is based on the decomposition of input objects into 1D atoms, thus taking benefit from the highly parallel nature of GPUs. Chapter 5 proposed an extension to perform internal context operations in parallel at the atom level. However, another problem remains for grammars constrained by external contexts: knowledge of the environment external to the input object. For instance, such grammars could be constrained to grow on an underlying surface, or be controlled with high-levels editing tools. Related works have been presented in Chapter 3. First, various ad-hoc methods have been developed to control the global structure of context-free grammars. These grammars grow freely in the 3D space, without any external context information. Using high-levels tools such as strokes [IOI06], guides [BŠMM11], paints or voxels [TLL+11], artists are able to force the generated objects to fit a desired global structure. While these methods only provide structural constraints, a more general approach consists in writing context sensitive grammars [Cho56], that is directly integrate external contexts within grammar rules. Depending on external contexts, the grammar is derived accordingly. Context sensitive L-Systems have been used to interact with their environment such as general surfaces [PJM94, MP96], height fields [PM01] and curves [PMKL01]. However, all these methods only consider growth on the plane or 3D space but not on surfaces. Li *et al.* introduced field guided shape grammars [LBZ+11] where the user designs vector or tensor fields onto an underlying surface. Then, the external context represented by the fields is used directly at the rule level to control the grammar expansion onto the surface, and field values can be used by any rule. A common drawback of these techniques is they all requires from seconds to minutes to generate such controlled models.

Currently in our pipeline, each 1D atom grows freely in the 3D space, without any constraints nor information about the environment. This leads to a lack of control of the grammar for the user and global consistency issues, such as inter-penetration, preventing coherent grammar generation. Basically, context sensitive objects would need to know at any time the position of the current scope relatively to the environment. Then, grammar rules could be controlled with queried information from external contexts: either surface or texture contexts. For instance an artist wishing to generate an ivy model following a statue mesh could use the surface context of the statue. In addition to surface contexts, one should be able to control the allowed growth places, the foliage density, the color of the leaves, etc, with simple texture contexts. We introduce a generalized approach where any external context may constrain any rule. While the grammar can follow an underlying surface using the corresponding surface context, we also use an artist-friendly painting solution for controlling grammar rules with texture contexts. Moreover, as interactive feedback is crucial for artist, our solution is adapted to the GPU shape grammars pipeline, thus benefiting from parallelism capabilities of the GPU.

Our solution integrates texture contexts (Section 6.2, Figure 6.2I) to constrain the grammar rules according to texture information. Texture contexts are either generated offline or painted on-the-fly. A texture lookup accessor is added within the grammar

in order to query texture contexts during rule expansion (Figure 6.2a). Our system also supports surface contexts based on *indirection geometry image* (Section 6.3, Figure 6.2II), a multi-charts geometry image with indirection information for jumping from a chart to another at run-time. Surface context generation is performed on the GPU from a parametrized input geometry (see Figure 6.2b to d). In Section 6.4, we detail a *marching* rule responsible for moving over the surface context using its indirection geometry image (see Figure 6.2e). The geometry mapping onto the elements generated by the marching rule is then performed using Bezier curve interpolation (Section 6.5, Figure 6.2f). Finally, we demonstrate that our method is able to generate many consistent growing models, such as ivy, in parallel using the GPU (Section 6.6). Thanks to these contributions, artists can control the grammar rules at different levels with a simple painting interface. Issues and future works are finally discussed in Section 6.7.

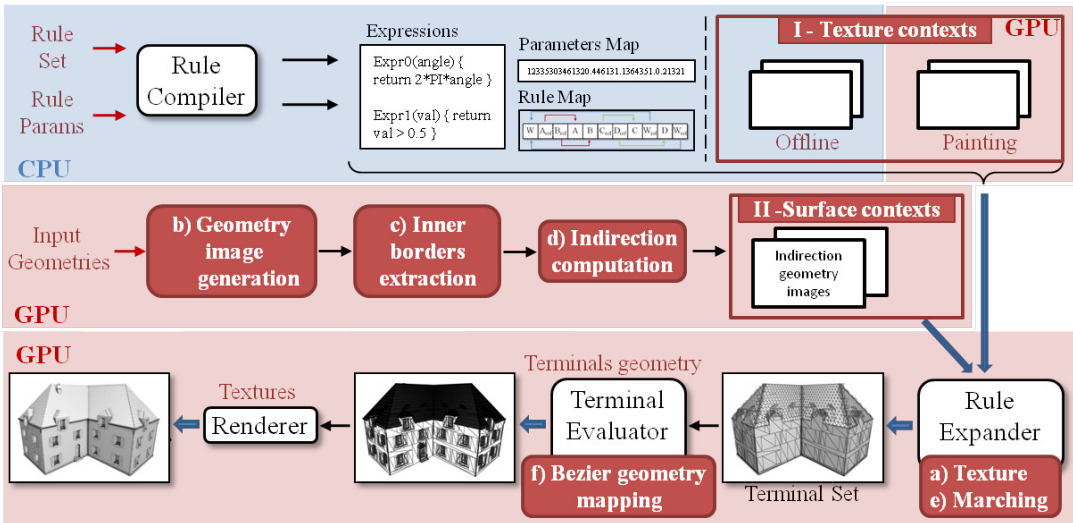


Figure 6.2: GPU shape grammars pipeline is extended to handle external texture (I) and surface contexts (II). Texture information is queried using a texture lookup accessor (a). Surface context is encoded into an indirection geometry image (b to d) and accessing surface information is performed by the marching rule (e). Finally, the terminal geometries are seamlessly synthesized over the terminal generated by marching rule (f).

6.2 Passing context through texture maps

A simple approach for controlling grammars with external contexts is to use textures (Figure 6.2I). Any information can be encoded into textures: a population map, a density map, a random map, etc. For instance, one may control the fur repartition on an object using a texture. In this case, the external context is the texture context. Starting from a tessellated mesh with given texture coordinates, each polygon of the mesh may be processed using our parallel pipeline. The context-sensitive grammar populating the mesh with fur is then governed by the population map from the user.

However, the user does not have any grammar tool to access texture information. While it would be very useful to constrain any rule with such information, adding a new special rule certainly complicates the work of the grammar designer. Instead, we introduce a texture accessor (see Figure 6.2a). Using an accessor directly as rule parameter allows for a cleaner grammar and faster execution. The texture lookup accessor fetches a texel of a given texture according to provided texture coordinates. As the rule parameters are automatically extracted into shader code expression (see Chapter 4), this accessor is defined as a classical texture lookup method in shader:

vec4 texture(int textureId, vec2 texCoord)

where textureId is the id of the texture to address, and texCoord are the texture coordinates corresponding to the pixel to fetch.

With such texture accessor, one may easily access texture context to constrain its grammar. The grammar hereafter populates a terrain according to the value of the population map in Figure 6.3.

GenBuilding → Condition(texture(0, getTexCoord()).x > 0.5)
{ StartBuilding, GenTree }
GenTree → Condition(texture(0, getTexCoord()).y > 0.5)
{ StartTree, NULL }

For each polygon of the terrain, the grammar evaluates the texture context according to the scope texture coordinates. For this grammar, a building is generated if the x value of the fetched pixel is upper than 0.5. Else, if the y value is upper than 0.5, a tree is then generated.

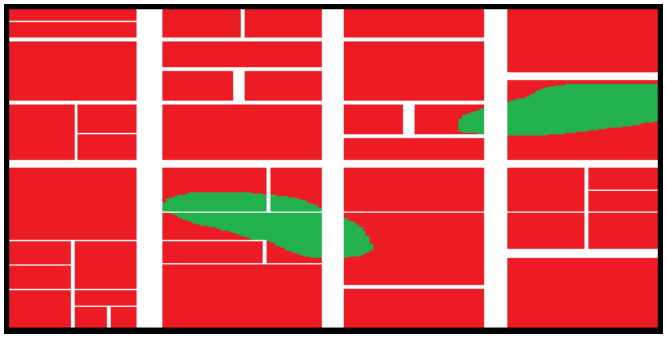


Figure 6.3: Population map of buildings and trees. Red pixels indicate a building area while green pixels specify trees. White pixels correspond to null terminal symbol.

The artist may feed the system with an existing population texture. In addition, as our pipeline provides interactive feedback, the artist can also paint the texture on-the-fly directly onto the model. The painted texture is generated on the GPU. While the artist paints onto the model, the texture coordinates corresponding to the brush center are sent to a paint shader. Then, depending on the brush size, the shader writes the texture map at the given texture coordinates with the brush value.

While this approach allows external texture contexts to constrain the grammar, generated models still grow freely into the environment. The following section describes a solution to external surface context in order to generate consistent growth over surfaces.

6.3 Indirection geometry images

In order to constrain the grammar expansion with information about its underlying surface, each 1D atom should know its position relatively to this surface context. Grammar rules should thus query geometric and texturing information all over the surface to guide themselves at any time. As the grammar evaluation stage is performed at run-time on the GPU, the surface context must be available directly on the GPU. Our approach uses texture-based mesh representation that is efficient for the GPU.

6.3.1 Geometry images

Various mesh data structure have been developed and optimized for a GPU usage. Data structures such as half-edge data representation [Män88] aim at efficiently encoding triangles information. While such a conservative approach permits to store every triangle of the mesh, it is also more complex to generate and to use than texture-based approaches. In this latter category of data structures, mesh information is encoded as a classical texture, that is easily accessed on the GPU using shaders.

Among texture-based methods, Gu *et al.* introduced geometry images [GGH02]. Starting from a mesh, a surface parametrization is first computed so that the mesh is topologically equivalent to a disk. A geometry image is then generated by rendering the mesh into a texture using the uv parametrization instead of vertices positions. The result is a flat textured representation of the mesh where each pixel contains one interpolated vertex position value. Other information such as normal and texture coordinates may also be stored as per pixel values. This geometry image can then be used to approximately reconstruct the initial mesh. While the quality of the reconstruction heavily depends on the texture size, it may also suffer from distortion artifacts due to the automatic sampling from the rasterization. Depending on the mesh parametrization, high curvature areas may be undersampled, or flat ones oversampled. Such sampling ends up with a low quality mesh representation leading to inconsistent mesh reconstruction from the geometry image.

Sanders *et al.* extended this work to multi-charts geometry images [SWG+03]. In order to limit these distortion artifacts, multiple cuts are found over the mesh to extract various charts parametrized independently. The charts are then distributed all over the texture through a chart placement algorithm. Finally, the classical geometry images rendering is applied in a first pass to generate the multi-charts geometry image. As rasterization on the border of the two adjacent charts may give different values on each chart, they apply a border zippering algorithm. Their algorithm avoids cracks and ensure continuity at chart borders. In a second pass, they create a one-pixel large border around each chart to force neighboring charts to share the exact same border. Better reconstructions are obtained, minimizing distortion artifacts.

Geometry images have been used for efficient ray-tracing of meshes on the GPU [CHCH06], proving this structure suitable for the GPU. However, while the reconstruction based on multi-chart geometry image is seamless due to the border zippering, each chart of the geometry image is independent from its neighbors. This representation does not allow to jump from a chart to another at run-time. For instance, being on border pixel of a source chart of the geometry image, there is no solution to find its neighbor in a destination chart. The following section introduces indirection geometry image in order to allow jumping from a chart to another on multi-chart geometry images.

6.3.2 Indirection computation

The surface context of the mesh is encoded into an indirection geometry image (see Figure 6.2II). It is a texture similar to multi-chart geometry images, with additional information for jumping from a chart to its neighbors. We only consider final mesh representation into texture atlases, but not the mesh parametrization issues. Existing multi-chart unwrapping algorithms [SPR06] are used to compute the mesh parametrization offline. From such atlases, the indirection geometry image generation algorithm automatically finds chart correspondences on the GPU.

An indirection geometry image is a texture atlas containing information about the mesh (*e.g.*, vertices position, normal, texture coordinates) with indirection information at chart borders. Figure 6.4 illustrates these indirections. We start from a mesh parametrized in a preprocessing step. As we compute a one pixel border containing indirection information around each chart of the texture atlas, the indirection geometry image needs at least two pixels between each chart. Similarly to geometry images, the texture resolution directly impacts the sampling quality of the mesh.

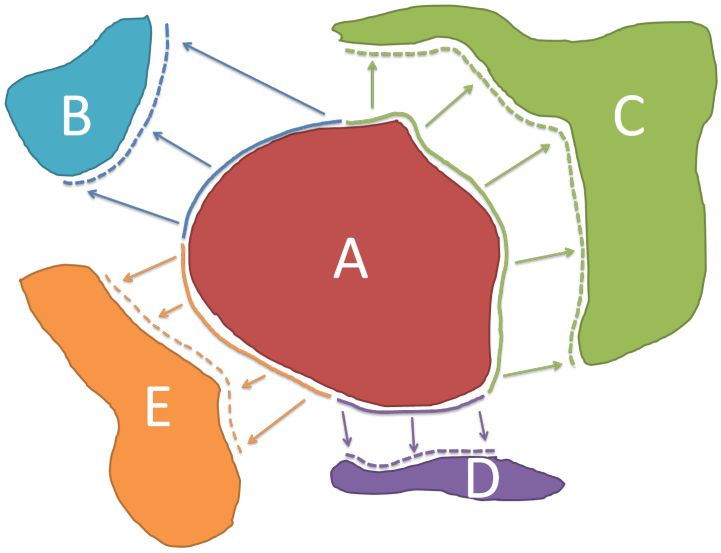


Figure 6.4: The indirection geometry image contains indirection information at chart border. The chart A is linked to the charts B, C D and E with indirections.

The indirection geometry image is composed of four types of pixels (see Figure 6.5). Pixels belonging to the geometry image are divided into two sub-categories (1 & 2):

1. pixels bordering the background are called inner border pixels
2. the others are inner pixels

The pixels being on the background are also divided into two sub-categories (3 & 4) :

3. pixels bordering the geometry image are outer border pixels or indirection pixels
4. the others are background pixels

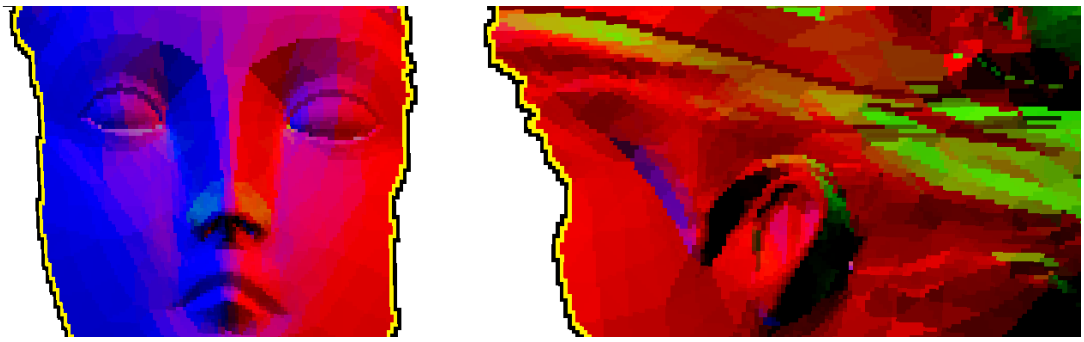


Figure 6.5: Inner border pixels (yellow) belong to the mesh, while outer border pixels (black) are added into the background of the indirection geometry image (white).

Our algorithm generates the indirection geometry image using three GPU-based stages. First, a classical multi-chart geometry image without border zippering operation is created (see Figure 6.2b). The principle is to render the input mesh onto a full-screen quadrilateral by inverting vertex positions and texture coordinates [GGH02]. The next stage is designed to isolate inner border pixels into a GPU buffer (Figure 6.2c). As the general algorithm finds the closest inner border pixel for each outer border pixel, regrouping all the inner border pixel into a unique GPU buffer considerably accelerate indirection computing.

Finally the last stage computes indirection for each outer border pixel (Figure 6.2d). It finds out which inner border pixel in another chart is the nearest to the indirection pixel (see Algorithm 3).

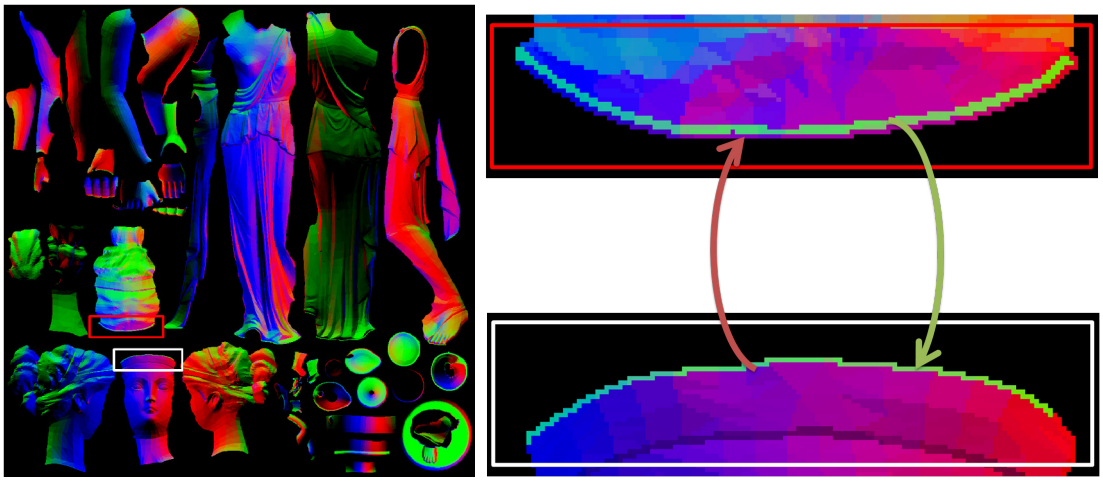
During indirection algorithm, charts id are compared so that the indirected pixel does not belong to the same chart. These charts id are computed offline during the multi-chart geometry image generation and provided as vertex attributes. An example of indirection geometry image is shown in Figure 6.6. Once the indirection geometry image corresponding to a 3D model is computed, grammar expansion may use the *marching* rule to follow the surface context.

```
// init
geometryImage ← input geometry image
innerBorderBuffer ← innerBorderBuffer generated
indirectionGeometryImage ← input geometry image
// algo
Render a full-screen quadrilateral
for all fragments (i, j) do
  tmpPixel ← geometryImage[i, j]
  if tmpPixel == background then
    isOuterBorderPixel = false
    mean ← 0
    // compute the mean of all inner neighbor pixel
    for k = −1..1, l = −1..1 do
      if geometryImage[i + k, j + l] != background then
        mean ← mean + geometryImage[i + k, j + l]
        isOuterBorderPixel = true
      end if
    end for
    if isOuterBorderPixel == true then
      // find the nearest vertex to the mean
      nearestVal ← +inf; nearestInt ← 0
      for k = 0..innerBorderBuffer.size-1 do
        if innerBorderBuffer[k].chartId != tmpPixel.chartId then
          if innerBorderBuffer[k] - mean < nearestVal then
            nearestVal ← (innerBorderBuffer[k] - mean)
            nearestInt ← k
          end if
        end if
      end for
      indirectionGeometryImage[i,j] ← innerBorderBuffer[nearestInt]
    end if
  end if
end for
```

Alg 3: Indirections are computed for outer border pixels. The reference value is the mean of the 1-ring neighborhood of the outer border pixel. Then the indirection corresponds to the closest inner border pixel from the reference value.

6.4 Marching rule

The *Marching* rule is responsible for generating extrusions along the surface. Contrary to the *Extrude* rule that perform extrusion anywhere into the 3D space, *Marching* rule performs extrusion following the surface context in any direction. The *Marching* rule is



(a) indirection geometry image (b) zoom on the head parts

Figure 6.6: Indirections are computed for each outer border pixel, referencing an inner border pixel from another chart a). Bottom part of the head replicates green border pixels of the top part, and inversely with purple ones b).

specified as follows:

Pred→Marching(vec2 direction, float stepSize) {quadExtruded, segmentTop}

where *Pred* is the rule predecessor, *quadExtruded* is the rule successor applied onto the extruded quadrilateral, and *segmentTop* is the rule successor applied onto the translated segment. The parameter *direction* is the 2D vector indicating the direction for the marching inside the indirection geometry image, and stepSize corresponds to the marching size in pixels. The *direction* parameter is optional, and if not provided, the marching direction of the current scope is used. Initial scope has the direction $(0, 1)$. Moreover, the rotation rule is extended to rotate the marching direction of the current scope.

The interpretation of the *Marching* rule is done similarly to the other rules, at the rule expanding stage of the GPU shape grammars pipeline. Starting from a seed sampled from the base mesh, its initial position, normal and texture coordinates are known, using the center of the segment. The texture coordinates are the entry point into context surface represented by the indirection geometry image. Then, the 2D direction and marching step size (*i.e.*, grammar rule parameters) are evaluated at run-time.

The marching algorithm starts from a source texture coordinates, and fetches a destination texture coordinates according to the given marching direction. Four cases may happen during the marching according to the pixel fetched:

1. it belongs to the same chart of the indirection geometry image than the current scope (Figure 6.7A)

2. it directly reaches an outer border pixel (Figure 6.7B)

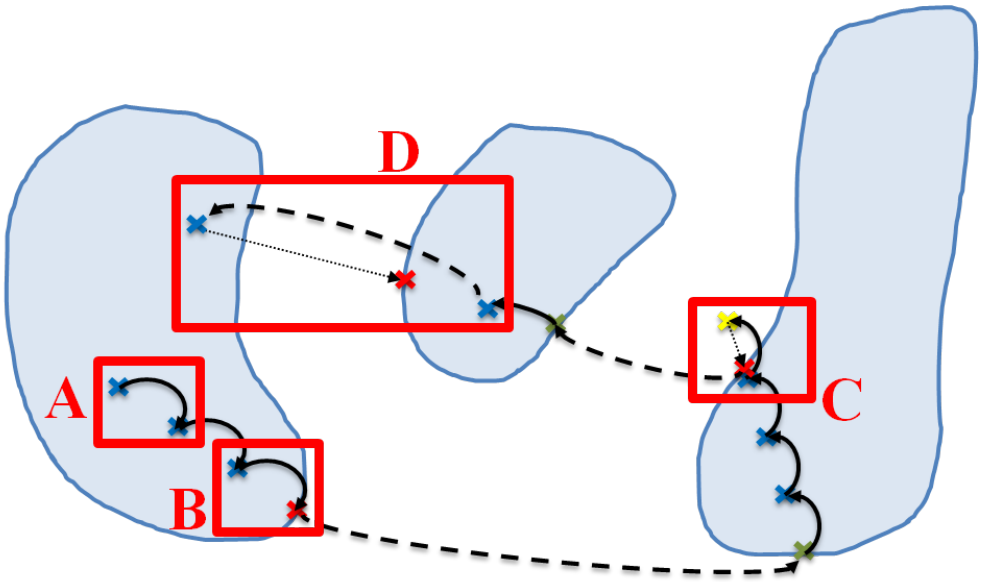


Figure 6.7: Four cases may happen during marching rule. Blue cross represent inner pixels, green cross are inner border pixels, red cross mean outer border pixels and yellow cross are background pixels. A) the pixel fetched is an inner pixel. B) the pixel fetched is an outer border pixel, the redirection is automatic (dash line). C) the pixel fetched is part of background, the marching is inverted to find the corresponding outer border pixel (dotted line). The redirection is then automatic. The case where the pixel fetched is part of another chart (*e.g.*, large marching step) is treated similarly to C).

3. it gets to the background (Figure 6.7C)

4. it ends directly into another chart (Figure 6.7D)

In the first case, the destination vertex simply corresponds to the pixel fetched. For the second case, as the outer border pixel contains the value of a vertex in another chart, the pixel fetched automatically redirects to this chart. Finally, if the pixel fetched jumps in the background or in another chart, the algorithm finds the outer border pixel in the inverse given marching direction. Then the redirection is performed as in the second case.

As the redirections are computed according to the mean of the inner border pixels, the outer border pixel fetched may not be the more appropriate. In order to fetch the outer border pixel minimizing the 3D direction distortion, a best matching pixel selection is performed at redirection (Section 6.4.1). Then, the 2D marching direction for the destination chart is computed to be consistent with the one of the source chart (Section 6.4.2). Finally, instead of generating classical terminal quadrilaterals, the marching rule generate quadrileterals over the mesh with specific tangent, binormal and normal at center endpoints (Section 6.5). Those information are later used during the terminal evaluation stage to allow smooth transitions between terminal shapes.

6.4.1 Best matching pixel

The indirection geometry image indicates for a given outer border pixel the redirection into its original chart. Each indirection pixel is set as the nearest inner border pixel being on another chart, according to the mean of its 1-ring inner border pixel neighborhood. This is why a better indirection pixel may be selected than the one fetched on the 2D marching direction, with respect to the 3D direction (Figure 6.8). In order to find the most consistent outer border pixel, the matching step considers $outer^k$ containing both the fetched outer border pixel, and its 1-ring neighborhood being also indirection pixels (Figure 6.8: P_0 and P_1, P_2, P_3).

First, a reference 3D direction is computed between the current vertex position $curPos$ and the vertex position $innerPos$ at the inner border pixel P_{inner} being on the 2D marching direction. Then, for each indirection pixel $outer^k$, the algorithm computes the 3D directions between $curPos$ and the position of the indirected vertex positions $outerPos^k$. Finally, for each outer border pixel $outer^k$, a matching value is computed as:

$$\frac{innerPos - curPos}{|innerPos - curPos|} \cdot \frac{outerPos^k - curPos}{|outerPos^k - curPos|}$$

The indirection pixel having the closest matching value to 1 is then the best matching pixel for indirection. Once the indirection pixel is selected, the 2D marching direction at the destination chart is oriented according to the source and destination tangents.

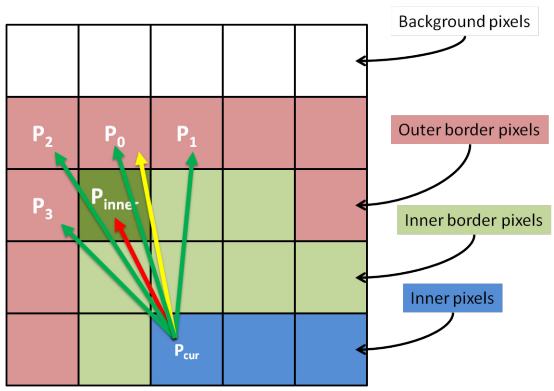


Figure 6.8: Marching from the pixel P_{cur} to outer border pixel P_0, the best indirection pixel may be in the 1-ring neighborhood of P_0. A reference 3D direction is computed according to P_{inner}, the closest inner border pixel from P_0 being on the 2D marching direction (yellow arrow). The reference value is the 3D vector $pos(P_{inner}) - pos(P_{cur})$ (red arrow). Then we compute the 3D direction vectors $pos(P_k) - pos(P_{cur})$ (green arrows). The outer border pixel whose 3D direction vector is the closest to the reference one is chosen as indirection pixel.

6.4.2 Tangent-based orientation of the marching direction

Adjacent charts may have various orientation. From this observation, the same marching direction from a chart A to a chart B can be used only if their adjacent edges are parallel. However, as soon as different rotations are applied to adjacent charts, a new orientation for the marching direction should be computed, when jumping from the chart A to B. Thus, being on a chart A with a given 2D direction, the consistent direction at chart B is a rotation of the initial direction (see Figure 6.9).

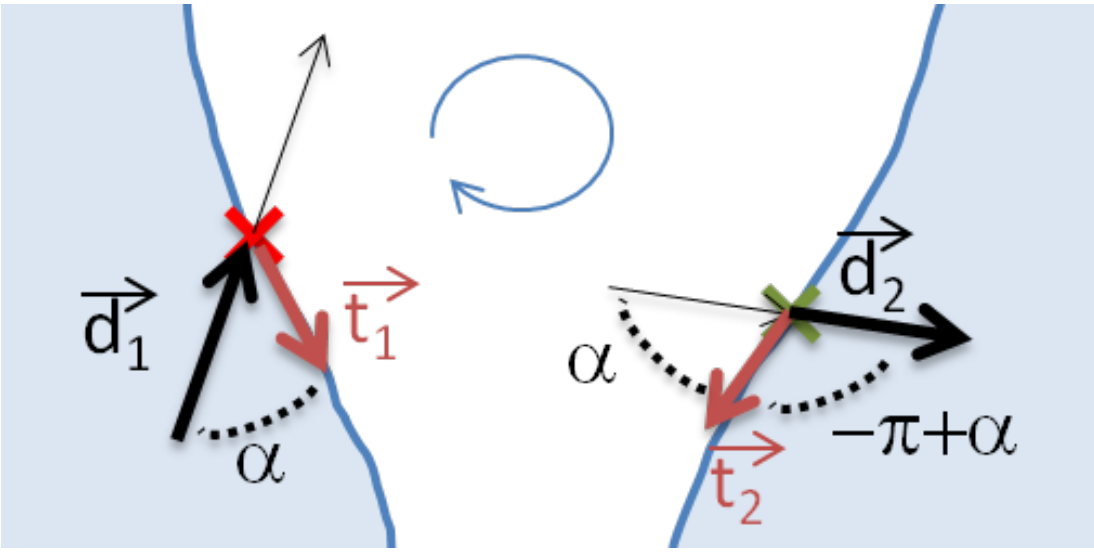
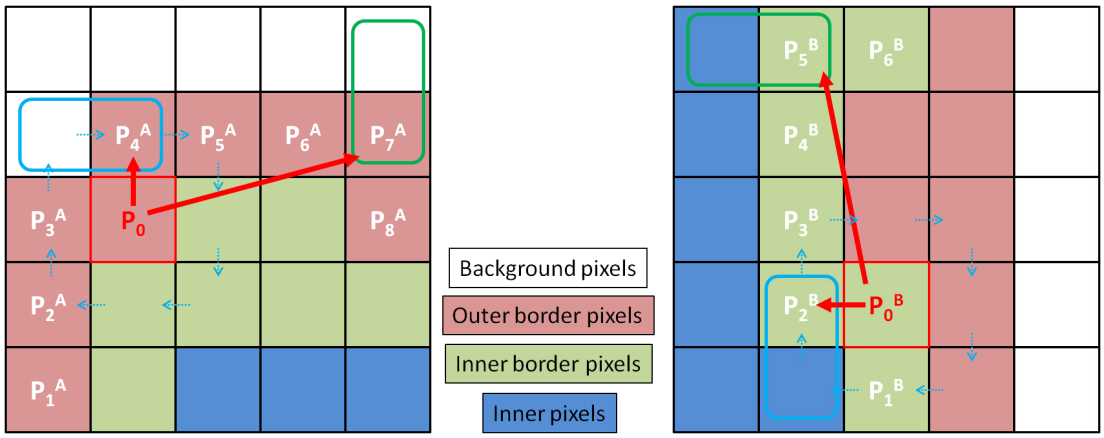


Figure 6.9: Marching direction is oriented according to the source and destination 2D tangent vectors. Let α be the angle between $\vec{t_1}$ and $-\vec{d}_1$, the destination marching direction d_2 is the rotation of t_2 of $-\pi + \alpha$.

In order to compute the rotation to apply to the 2D marching direction, local tangents at the border of the source and destination charts are computed. Our solution is invariant to rotation, translation and scale operations computed during chart placement step of the parametrization, but not to mirroring operations. As this rotation algorithm is based on the tangent of each charts, both charts must be consistent in orientation. That is, they both have to be oriented either clockwise or counter-clockwise. If adjacent charts are not consistently oriented, the algorithm will fail and inverse the rotation.

First, the tangent for the source chart A is approximated (see Figure 6.10a). Being on an outer border pixel of A, with texture coordinates *nextTexCoord*, the 1-ring neighborhood is scanned in clockwise order. During this scan, the first outer border pixel after a background one, with texture coordinates *nextTexCoordTangent*, is part of the source tangent. This tangent is thus defined as *nextTexCoordTangent* − *nextTexCoord*. The angle α betwen the source tangent and the source 2D marching direction is computed. Then the outer border pixel at *nextTexCoord* redirects to the inner border pixel of destination chart B, at *newTexCoord* (Figure 6.10b). The 1-ring neighborhood of this inner border pixel is scanned. Here, the first inner border pixel



(a) Source chart A. (b) Destination chart B.

Figure 6.10: Chart tangent computation at one and three pixels distance, respectively blue and green case. In the source chart, the tangent endpoint is defined as the first outer border pixel after a background pixel. For the destination chart, the tangent endpoint is defined as the first inner border pixel after an inner pixel. Computed tangents are indicated in red.

after an inner pixel, with texture coordinates $newTexCoordTangent$, is part of the tangent at chart B. The destination tangent is thus defined as $newTexCoordTangent - newTexCoord$. Finally, the destination 2D marching direction is a rotation of the destination tangent of $-\pi + \alpha$.

However, fetching only the 1-ring neighborhood to compute the tangent may give inconsistent results. For instance, if two adjacent chart borders are rasterized sightly differently, the rotation may be over-estimated. Better tangent estimation is provided by computing the tangent at various pixel distances. The final tangent computation is modulated with distances of one pixel, three pixels and five pixels border tangents. Mean of the intermediate tangents allows smoother tangent estimation. A future work could be to experiment with the tangent approximation based on a discrete curve [FT99].

The *marching* rule provides a simple way to move according to the context surface using an indirection geometry image. The best matching pixel recovers the more consistent pixel at border. Jumping from a source chart to a destination chart is automatic using the texture coordinates of outer border pixels. Then the destination 2D marching direction is rotated according to the angle between the source tangent and source 2D marching direction. Finally, quadrilaterals are generated between marching endpoints with normal and tangent information. Those segments are then used for smooth seamless geometry mapping.

6.5 Smooth seamless geometry synthesis

In order to substitute a terminal symbol by geometry, artists commonly use the *Shape* rule. It indicates the terminal geometry id to instantiate at the terminal evaluation stage. In the GPU shape grammars pipeline presented in Chapter 4, terminal evaluation considers only geometry mapping on flat quadrilaterals. Using a simple linear mapping based on the position of the four corners, the terminal shape fits the quadrilateral support. However, in case of multiple extrude rules that do not follow a unique extrusion direction, each instantiated terminal shapes is oriented according to the normal of its quadrilateral, thus exhibiting a continuity issue (see Figure 6.11a).

Vlachos *et al.* introduced PN-triangles for generating smooth subdivided triangles with a C^0 continuity and C^1 at triangle corners [VPBM01]. As PN-triangles only need vertex position and normal inputs, it is easily integrable into graphics hardware applications. A PN-triangle is defined as a cubic triangular Bezier patch that matches the point and normal information at the vertices of the flat triangle. Peters [Pet08] extended this work to PN-quads. While we could use PN-quads for continuous geometry synthesis over the quadrilateral terminal shapes, this solution requires to compute 12 control points and 5 control normals over the quadrilateral patch, and then to interpolate the terminal shape vertices accordingly. To prevent from complex operations at terminal evaluation stage, simpler cubic Bezier curves are proposed.

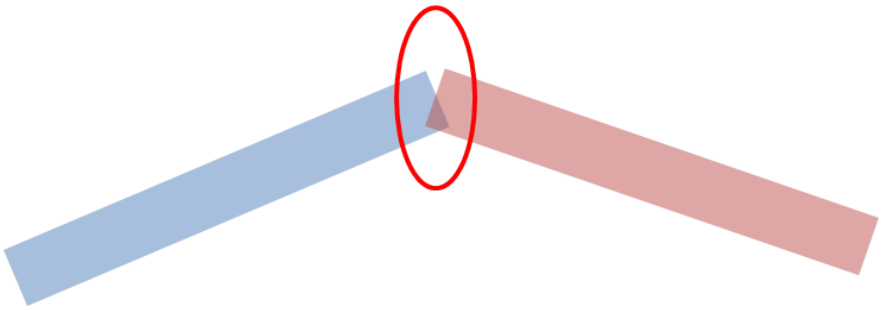
Instead of linear mapping of the terminal geometry over the quadrilateral, we consider straight lines joining center endpoints of the quadrilateral, with position, normal and tangent information. While position and normal are directly queried from the in-direction geometry image, the tangent is computed as the cross product between the mean adjacent binormal and the normal of the point. Given three points P_{i-1}, P_i, and P_{i+1}, and N_i being the normal at P_i, the tangent T_i at P_i is:

$$T_i = N_i \times (\frac{1}{2}(\frac{P_{i+1} - P_i}{||P_{i+1} - P_i||} + \frac{P_i - P_{i-1}}{||P_i - P_{i-1}||})) \tag{6.1}$$

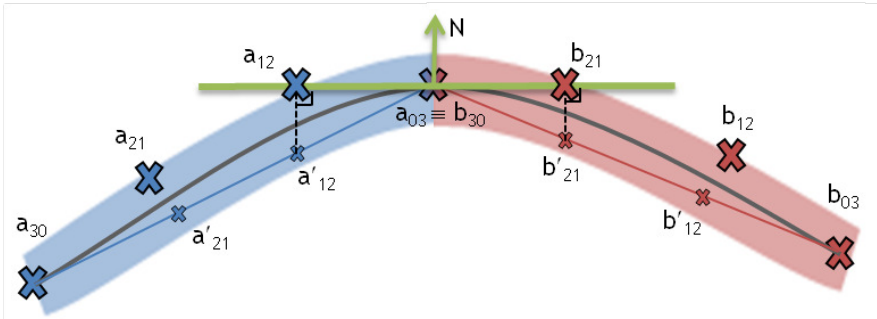
where $\times$ defines the cross product.

In order to apply a cubic Bezier interpolation for a straight line defined by its two endpoints, we need to compute two more control points. Following the idea of Vlachos *et al.*, control points of the Bezier curve are computed according to the projection normals [VPBM01] (see Figure 6.11b). First two control points uniformly sample the straight line at $\frac{1}{3}$ and $\frac{2}{3}$. Then for each endpoint, the nearest control point is projected onto the plane defined by the projection normal N'_i. By setting projection normal N'_i in the plane defined by P_{i-1}, P_i, P_{i+1}, we ensure G^1 continuity. Finally, the two control points are defined according to the projection normal as follows:

$$\begin{aligned} &b_{30} = P_i, b_{03} = P_{i+1} \\ &w_{ij} = (P_j - P_i) \cdot N'_i \in \mathbb{R} \\ &b_{21} = (2P_i + P_{i+1} - w_{i,i+1}N'_i)/3, \\ &b_{12} = (2P_{i+1} + P_i - w_{i+1,i}N'_{i+1})/3 \end{aligned} \tag{6.2}$$



(a) Linear mapping artifacts



(b) b_{21} control point computation

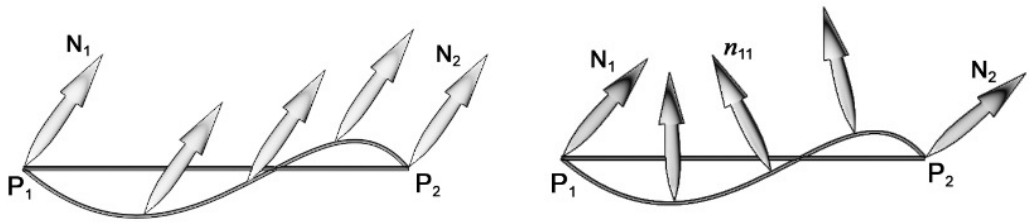
Figure 6.11: Multiple extrusions with various extrude directions prevent from seamless transition (a). Using the projection normal and position information of the segment defined by the endpoints b_{30} and b_{03}, we define two control points b_{21} and b_{12} for cubic Bezier interpolation (b). b_{21} is the projection of $(2b_{30} + b_{03})/3$ into the tangent plane at b_{30}. b_{12} is the projection of $(b_{30} + 2b_{03})/3$ into the tangent plane at b_{03}.

The Bezier curve is then computed using the equation:

$$B(t) = (1-t)^3 b_{30} + 3(1-t)^2 t b_{21} + 3(1-t)t^2 b_{12} + t^3 b_{03} \tag{6.3}$$

By defining a normal point at mid-edge, a quadratic normal interpolation captures shape variations (*e.g.*, inflexions), which is not possible with linear interpolation as illustrated in Figure 6.12. This mid-edge normal point n_{11} is defined as the average of the end-normals reflected across the plane perpendicular to the segment:

$$\begin{aligned}
n_{20} &= N_0 \\
n_{02} &= N_1 \\
n_{11} &= \frac{h_{11}}{||h_{11}||} \\
h_{11} &= N_0 + N_1 - v_{01}(P_1 - P_0) \\
v_{01} &= 2\frac{(P_1 - P_0)\cdot(N_0 + N_1)}{(P_1 - P_0)\cdot(P_1 - P_0)}
\end{aligned} \tag{6.4}$$



(a) Linear interpolation. (b) Quadratic interpolation

Figure 6.12: Linear interpolation of the normal endpoints (a) does not preserve the inflexion of the surface, while quadratic interpolation does. Figures courtesy of [VPBM01].

The final quadratic equation for interpolating normal endpoints is:

$$N(t) = (1-t)^2 n_{20} + 2(1-t)t n_{11} + t^2 n_{02} \tag{6.5}$$

Tangent endpoints are also interpolated using the same method used for the normal. Finally, we only compute two control points, one normal point and one tangent point before interpolating the terminal geometry.

Geometry synthesis onto each generated quadrilateral is then performed in parallel using interpolated Bezier information: position, normal and tangent. Let $p_s(x_s, y_s, z_s)$ be a vertex of the terminal shape, where x_s, y_s, z_s are normalized (*i.e.*, relatively to the bounding box of the terminal shape, see Figure 6.13a). Computation of $p(x, y, z)$, the interpolation of p_s according to the Bezier curve, is performed incrementally. First, p is positioned according to the Bezier curve interpolation using its height value y_s (Figure 6.13b) as:

$$p = B(y_s)$$

Then, p is translated according to the quadratically computed tangent at y_s using its width value x_s (Figure 6.13c) as:

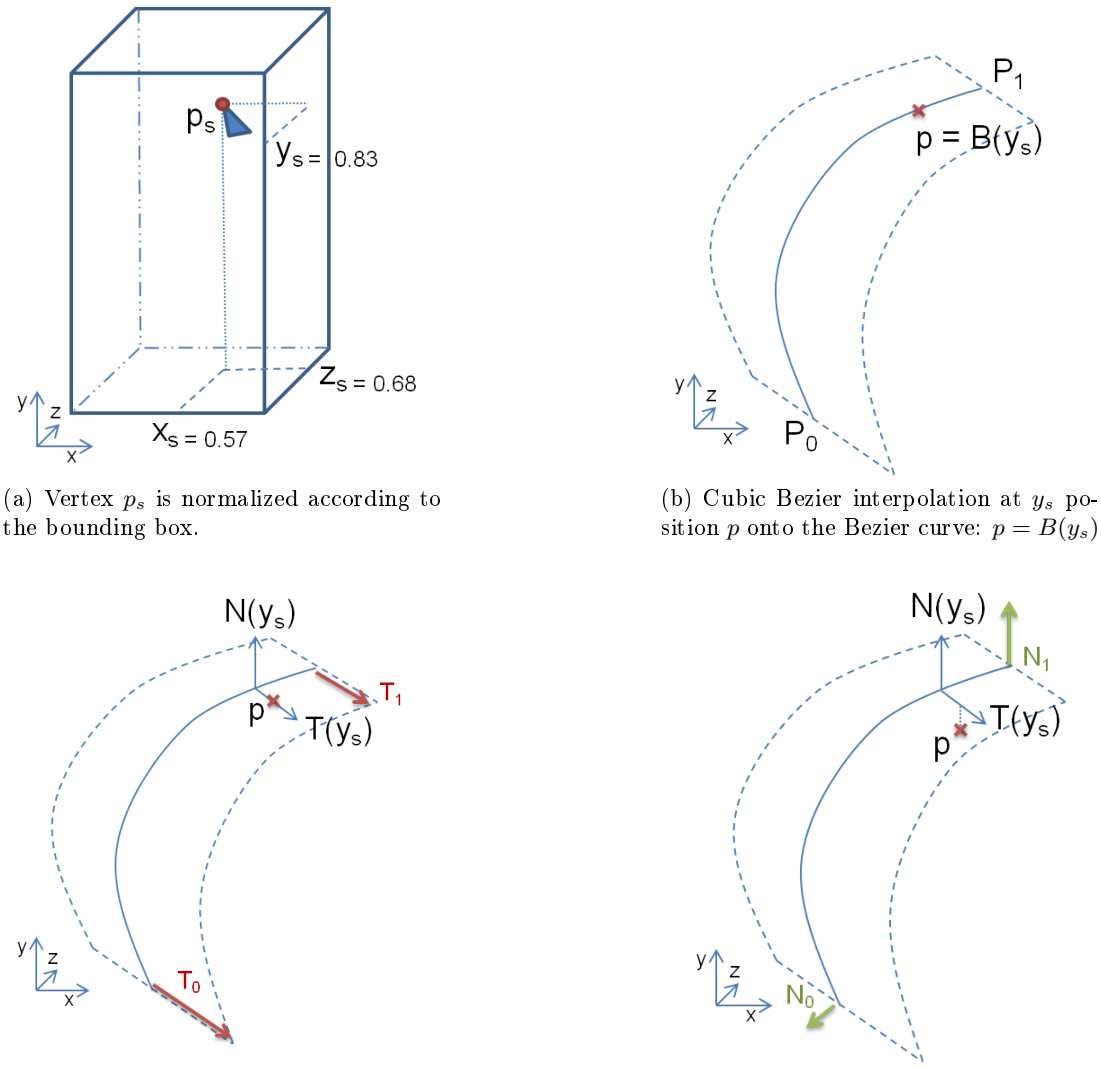
$$p = p + T(y_s)(T_0(1-y_s) + T_1 y_s)(x_s - 0.5)$$

where T_0 and T_1 are respectively the length of the tangent at endpoints P_0 and P_1. Finally, p is translated according to the interpolated normal at y_s using its depth value z_s (see Figure 6.13d):

$$p = p + N(y_s)(N_0(1-y_s) + N_1 y_s)(z_s - 0.5)$$

where N_0 and N_1 are the length of the normal at endpoints.

In the end, terminal shapes are smoothly interpolated according to the Bezier curve. Considering unique normal and tangent information for adjacent quadrilateral generated with the marching rule, repetitive terminal shape models are seamlessly generated at terminal evaluation.



(a) Vertex p_s is normalized according to the bounding box.

(b) Cubic Bezier interpolation at y_s position p onto the Bezier curve: $p = B(y_s)$

(c) p is translated according to the interpolated tangent at y_s and its width x_s: $p = p + T(y_s)(T_0(1-y_s)+T_1 y_s)(x_s - 0.5)$

(d) Finally p is translated according to the interpolated normal at y_s and its depth z_s: $p = p + N(y_s)(N_0(1-y_s) + N_1 y_s)(z_x - 0.5)$

Figure 6.13: Incremental computation of the position p according to the Bezier curve.

6.6 Applications & results

In this section we apply our method for generating external context-sensitive grammar-based models such as ivy plants growing onto a mesh. We also illustrate the expansion process controlled with various texture contexts. All the images are generated using a NVIDIA GeForce 580 GTX in a 1280 × 720 resolution.

6.6.1 Ivy growth

While the approach in Chapter 4 only allows to create plant models freely in the 3D space, the external surface context extension permits to restrict the growth process to follow any underlying geometry. For instance, ivy plants growing onto a base mesh are generated with our system.

In this example, the underlying surface that is partially covered by ivy is the Hebe model (Figure 6.14). This geometry comprises approximately 64,000 triangles. The mesh parametrization is computed in a pre-processing step, resulting in a multi-chart texture atlas. Then, the indirection algorithm generates the equivalent indirection geometry image in 2.1 seconds for a 1024 $\times$ 1024 resolution (Figure 6.6a). From this texture, the grammar-based expansion is performed onto the Hebe model at rule expanding stage from multiple seeds.

Figure 6.14: Our ivy model grows freely onto the Hebe model from 16 seeds. The grammar used is described hereafter, with $stepNumber = 30$, $stepSize = 12$, $stepBranch = 13$ and $numberOfLeaves = 2$. The rule expander takes 61 ms while terminal evaluation and rendering is performed in 11 ms. This scene represents 572K generated faces.

The ivy grammar follows the growth process described hereafter. The first marching rule initializes the marching direction while the other uses the direction of the current

scope, depending on the *stepSize* parameter. At every *stepBranch* marching step, a new branch is created and both branches rotate the marching direction of the current scope using the *RotateDir* rule. The ivy leaves are generated all along branches using a *Sampling* rule. This rule randomly samples the current scope and creates a given number of children. Finally, the ivy branch is an instantiated cylinder model of 216 polygons, for each extruded marching quadrilateral, while the leaves are simple textured quadrilateral.

```
IvyInitMarching        →  Marching(dir, stepSize)
                              { IvyDuplicate, IvyCondMarching(0)}
IvyCondMarching(n)     →  Condition(n < stepNumber )
                              { IvyBranching(n+1), NULL }
IvyBranching(n)        →  Condition( mod(n, stepBranch) == 0 )
                              { IvyBranch(n), IvyMarching(n) }
IvyMarching(n)         →  Marching( stepSize )
                              {IvyDuplicate, IvyCondMarching(n)}
IvyBranch(n)           →  Branch(2) { IvyRLeft(n), IvyRRight(n) }
IvyRLeft(n)            →  RotateDir( -angleRotation, 1 )
                              {IvyCondMarching(n)}
IvyRRight(n)           →  RotateDir( angleRotation, 1 )
                              {IvyCondMarching(n)}
IvyDuplicate           →  Branch(2) { CylinderShape, IvyPart }
IvyPart                →  Sampling(numberOfLeaves, 1.0) { IvyLeaf }
IvyLeaf                →  Extrude( 0.02, 5.0 ) { RegularQuad, NULL }
```

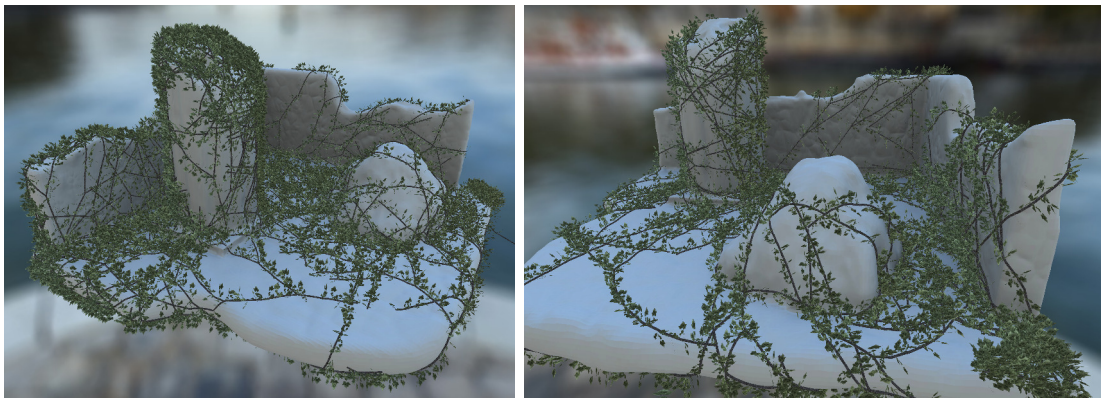
Figure 6.15 shows multiple ivy seeds expanded using the same ivy grammar, onto a different model. In this case the ruins model comprises 170K triangles, and the indirection geometry image is generated in 1.08 seconds.

Seeds are created at a picking step by the user directly onto the underlying model. Each time a user picks a new seed location, its position, normal and texture coordinates are read back. Then a new seed is added: the corresponding 1D atom is sent to the input seed buffer, with associated random parameters stored into the parameters map. The seed grammar is thus evaluated on-the-fly. As the initial growing direction is a grammar parameter, the user have to specify a value. We provide an intuitive way to set the first growing direction. To do so, the user indicates the initial growing direction for the seed by giving an impulsion at seed picking with a paint stroke. Let $startPickingTexCoord$ and $endPickingTexCoord$ be respectively the texture coordinates at picking press and release, the marching direction is then defined by:

$$endPickingTexCoord - startPickingTexCoord$$

However, this picking direction is only valid in cases where press and release belong to the same chart. Otherwise, the direction may be incoherent.

Using the GPU shape grammars pipeline, each seed is processed and the associated grammar is evaluated in parallel during rule expansion; the terminal sets are also evaluated in parallel at terminal evaluation. As an example, 1 to 30 seeds are generated at



(a) 421 ms / 87 ms (b) 194 ms / 41 ms

Figure 6.15: The ruins model is covered by 18 (a) and 24 (b) ivy plants, with grammar parameters $stepNumber = 34/48(a/b)$, $stepSize = 10/13$, $stepBranch = 6/14$ and $numberOfLeaves = 2/2$. Respectively 5.17M and 2.27M polygons are generated. Timings are indicated as rule expander / terminal evaluation and rendering.

the same cost for the ruins scenes in Figure 6.15. The sublinear complexities observed in Chapter 4 and Chapter 5 are also underlined in external context grammars. While this grammar describes a growing process anywhere of the surface, without additional external constraints, our method also provides the possibility to control the grammar expansion with such constraints.

6.6.2 Grammar constraints through texture contexts

The grammar presented above has one constraint guiding its expansion which is to always stay close from the underlying surface. In this case, the grammar expansion is constrained with the surface context of the model. However, one may want to add more external constraints to control the grammar.

Using the texture coordinates of the current marching scope, an artist can access any information at the defined position of the surface context in an additional texture. The information about the underlying surface may be geometric such as the curvature, or texturing. In this last case, artists paint values directly onto the mesh that are stored on-the-fly into a texture map using the texture coordinates on the mesh.

For instance, let us add a painting constraint to the $IvyCondMarching$ rule in previous grammar set. By changing the condition to:

$$n < nbStep \quad \&\& \quad texture2D(paintMap, texCoordCur).x > 0.5$$

one may specify that the ivy is able to grow to this surface position only if the painted texture has a red value greater than 0.5 at the defined location. Then, the artist simply paints onto the mesh the areas where the ivy should stop growing (see Figure 6.16).

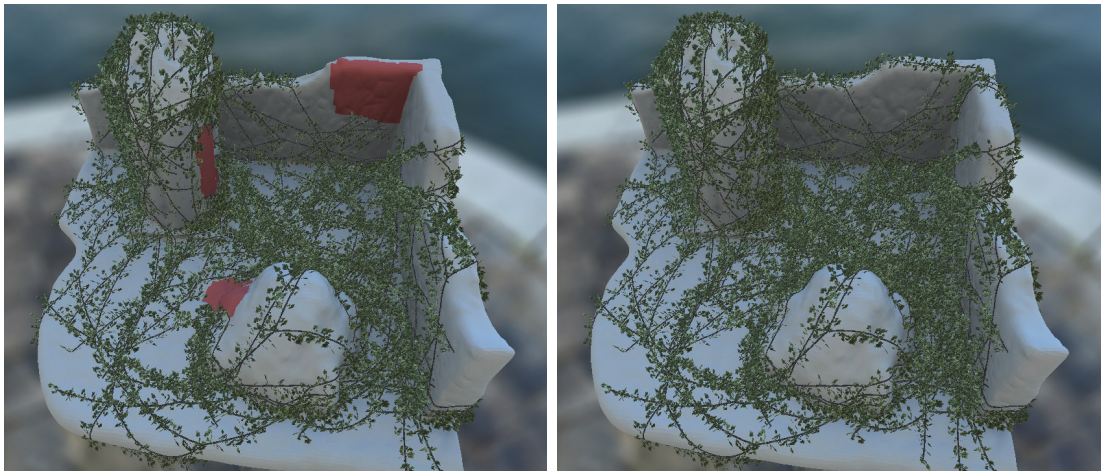


Figure 6.16: Texture context is used for interactively paint the restricted growing areas. Areas painted in red are forbidden to growth (left). The same ivy grammar grows freely onto the model without any restriction (right).

A second idea is to strictly guide where the grammar grows over the mesh. To achieve such control, multiple texture fetch associated with rotation rules may be used until an appropriate growing direction is found (Figure 6.17). Another solution would be to generate a vector field from the painting and use this value at a lower resolution to directly drive the marching direction. Various applications could constrain the grammar using the same approach, for instance to prioritize growth in cavity and modulate the ivy size on curved areas.

Figure 6.17: Texture context is also used to define strict growing areas. Ivy grammars follow the red painting on the mesh.

6.7 Discussion

Using the painting approach could be interesting in order to populate terrains with various grammars. For instance, one may paint on a terrain the areas where trees should be generated, other areas dedicated to the creation of buildings and so on. However, as the GPU shape grammars is composed of only one rule expanding stage, we have no other choice than regrouping both population grammar and object generation grammar altogether. It could thus be performed within the same generation pass. Nevertheless, this solution is not optimized because both grammars are decorrelated. They could be used separately in other situations. Ideally, a population grammar would generate multiple atoms in a first rule expander pass. Then, in a second pass, the rule expanding will take the generated segment as input and generate the corresponding geometries for each grammar. This would achieve better performances and benefit from the parallelism at both passes. In order to allow the use of multiple in/out buffers, multiple rule expanding stages should be performed. A solution would consist in describing the desired buffers on the grammar. This requires to write an expansion pipeline within the grammar, and to generate this pipeline on-the-fly.

While the marching rule provides expansion on a surface context, intersections may happen during the expansion process. Three types of intersection are defined: self-intersection where a grammar may generate many times a geometry at the same position, seed-intersection where various seeds may generate geometries at the same position, and surface-intersection where the generated geometry may intersect the underlying surface. Depending on the marching step size, highly convex areas may be intersected with the generated geometry. While decreasing the marching step size at intersected location reduces the intersection, the geometry synthesis algorithm based on Bezier curve interpolation also helps by taking care of the surface normal. To prevent seed-intersection and self-intersection an atomic counter can be used for each pixel of the texture. An atomic counter is a counter that may be incremented by all threads running in parallel. Thus, each time a marching expansion is performed, the atomic counter is incremented indicating the number of ivy already passed before. By modulating the distance to the surface according to this atomic counter, seed and self-intersections are limited. However, this solution tends to decrease a bit the parallelism speed. While the atomic counter is designed to limit the time for threads waiting access to the counter, it necessarily degrades parallelism performances.

6.8 Conclusion

This chapter presented a solution for controlling external context-sensitive grammars. A first approach is to constrain the grammar with texture contexts by allowing the grammar designer to use classical shader texture lookup methods as grammar accessors. Then, the texture accessor may be used within the grammar at any rule. While this approach provides the ability to control the rules at the initial seed position, another issue is raised concerning expansion process. In this case, any expansion scope should be able to access information about the underlying surface at any time. Using the *marching*

rule, one may constrain the grammar to be expanded according to a surface context. The indirection geometry image provides an efficient mean to encode the underlying surface with indirection between adjacents charts, allowing fast access on the GPU. Finally, as the grammar may be controlled with external information attached to the surface, artists may paint directly onto the surface to constrain the grammar with interactive feedback. Our solution thus provides an intuitive tool for external context-sensitive grammars.

Chapters 4 to 6 presented solutions for generation and rendering of highly detailed models using the GPU. Our pipeline handles up to hundreds of input 1D atoms at interactive time taking advantage of the parallel architecture of the GPU. However, a bottleneck appears for large-scale environments containing thousands of objects. The next chapter introduces a solution for parallel grammar scalability, allowing generation and rendering of elements composing very large sceneries.

Chapter 7

Parallel grammars scalability
and application to massive sceneries

(a) 7.5 fps (b) 8.3 fps

Figure 7.1: Our massive management system integrated within the GPU shape grammars pipeline allows for interactive design, editing and navigation in massive environments. The geometry of this scene comprises 116573 buildings and 561280 trees.

We introduce a massive management system integrated within the GPU shape grammars pipeline. To this end we developed a scalable framework by introducing automatic generation of multiple levels of detail at reduced cost. We apply our solution to the interactive generation and rendering of massive scenes containing thousands of buildings and trees. This work has been published in the GPU shape grammars paper [MBG+12].

7.1 Introduction

GPU shape grammars pipeline is designed to generate and render multiple procedural objects in parallel using a batching approach (Chapter 4). For each input segment, we derive and interpret the associated grammar to obtain one or several fully detailed object. In order to create more structured grammar-based objects, we proposed in Chapters 5 and 6 to enrich the 1D atom contexts with internal and external scopes.

A new challenge arises when we want to address the generation of massive sceneries at interactive frame rate, for instance to create virtual cities (see Figure 7.1). While our GPU shape grammars pipeline as presented in Chapter 4 supports interactive generation and rendering of hundreds of objects, scaling up to thousands of inputs shows some limitations in terms of memory consumption and generation time. Figure 7.2 recalls the pipeline introduced in Chapter 4, with the two GPU-based geometry amplification stages: the rule expander and the terminal evaluator. If we recompute the rule expansion and the terminal evaluation every frame, we obtain a "costly" generation time. However, we allow intermediate storage for the two stages. Thus we could cache the fully detailed objects on graphics memory after each parameters modification. Therefore, we avoid running a per-frame generation step but it leads to a higher memory cost especially at storing final generated objects.

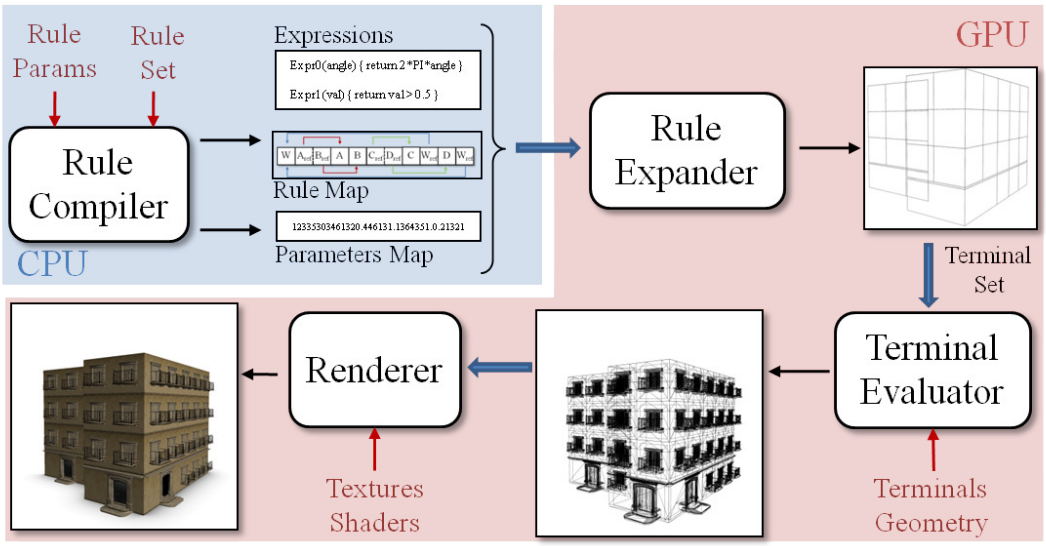


Figure 7.2: GPU shape grammars pipeline.

On the other hand, rendering so many detailed objects is hardly possible in real-time or even at interactive framerate. Typical scenes comprising thousands of highly detailed objects are challenging to render in real-time. To achieve such performance, level-of-details (LODs) and advanced rendering optimization techniques are mandatory.

In this chapter, we introduce a scalable approach based on automatic seamless LODs, integrated withing the GPU shape grammars pipeline. This provides interactive genera-

tion, as well as on-the-fly editing and rendering of objects composing massive sceneries. We will first see a fine-grain level-of-details system for grammar-based models using automatically generated image-based impostors (Section 7.2). We propose two different strategies for architecture and vegetation exploiting object structures (Sections 7.3 and 7.4). Finally we will talk about the caching, clustering and culling optimization strategies (Section 7.5).

7.2 LOD system overview

The output of procedural geometry generation is usually a highly detailed model. While rendering of one to hundred high definition model may be performed in real-time (see Chapters 4, 5 and 6), it is quite challenging to render in real-time an entire scene comprising thousands of such models. Basically it is not possible to exploit the massive power of procedural modeling directly into huge scenes: our parallel approach is not sufficient anymore. A solution consists in degrading the quality of generated objects, thus not reflecting the final rendering. In the case one cannot tolerate such a loss of quality, artists have to edit the parameters or the grammar rules outside of the scene on one model alone, then reintegrate the modification inside the complete scene for high quality off-line rendering. Thus modification of objects are very time-consuming for artist, acting in trial and errors process. This drawback prevents from interactive edition and feedback on massive grammar-based models. We introduce a dynamic LOD system for high quality real-time visualization (Figure 7.3).

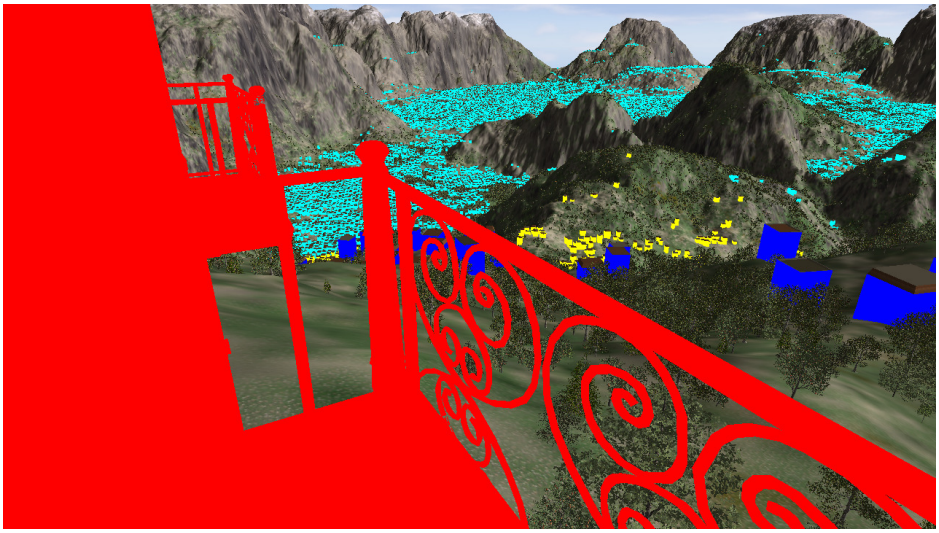
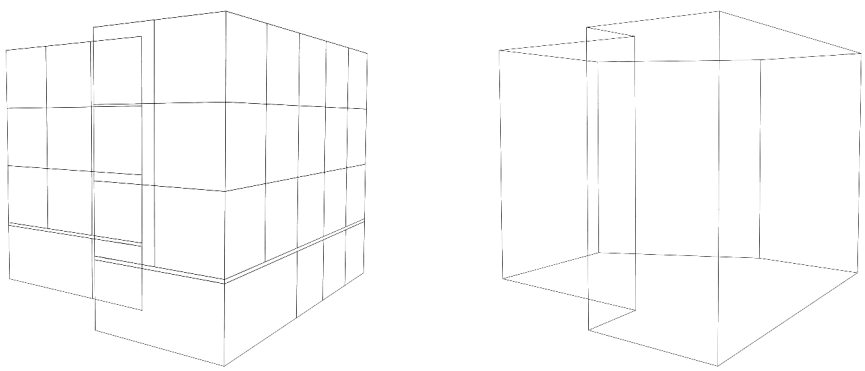


Figure 7.3: The management of levels of detail is integrated within our pipeline for interactive generation and visualization of large-scale sceneries. We defined five LODs for architecture (Figure 7.7): red buildings correspond to the highest level (LOD4), blue ones are central view impostors (LOD2), yellow ones correspond to grammar ray-tracing (LOD1) and cyan buildings are the lowest level simply using the wall texture (LOD0).

Our technique comes from a simple observation: in a massive scene such as the one in Figure 7.3, there are few objects closer to the camera which have a high screen coverage while the ones far away from the camera are much more numerous but cover less pixels. Due to the small screen coverage of the distant objects, we may accept a loss of quality for distant objects and degrade terminal shape geometries without impacting too much visual result. Moreover, in order to allow massive scenes composed of thousands of elements we also need to reduce the geometric load of the generated models. We thus define different LODs for architecture and vegetation, balancing rendering quality and memory storage. As buildings and trees follow two different grammar strategies, respectively reduction and growth processes, we developed accordingly different LOD strategies (Sections 7.3 and 7.4). While highest quality LOD always remains the actual geometric model, we substitute the geometry by image-based impostors for lower quality levels [JWP05]. Replacing thousands of polygons by a simple texture relieves the GPU from triangle processing cost. We first describe the LOD system developed for architecture modeling.

7.3 Architecture

We devise LODs for architecture based on the following observation: starting from a basic representation of the object, each step of the procedural generation refines the model and generates additional details to reach the best quality. Lower LODs can then be obtained by stopping the generation pipeline at chosen stages. Especially, for building models we often start performing one or a few extrude rules (growth) to generate the facades. Then many reduction rules (split and repeat) are applied to refine the facades and thus position block elements such as windows, doors, etc. Accordingly to this modeling process, we define two grammar-based LODs where the lower one do not perform entire grammar expansion (Figure 7.4).



(a) The higher quality LOD expands the overall grammar

(b) We stop the expansion at the first growth rule for generating the lower quality LOD

Figure 7.4: We define two grammar-based LODs for architecture modeling.

This lower level is called the extruded set. We either automatically stop after the first growth rule, or let the grammar designer tags the grammar to indicate the extrusion level corresponding to the extruded set. Using such a lower grammar-based LOD for the farthest objects, we limit the memory consumption for barely visible objects. On the other hand, the highest geometric LOD is simply the fully expanded one. Finally, we subdivide again the two grammar-based LODs to exhibit five terminal-shapes-based LODs. Our solution relies on intensively use of image-based impostors to avoid geometry generation for some of the LODs.

7.3.1 Impostors

A classical approach consists in generating a separate impostor for each object. However, procedurally-generated models tend to reuse a relatively small number of terminal shapes. Therefore, we generate per-terminal impostors, and combine them at render time. Such per-terminal impostors are a more compact solution than generating per-object impostors for each set of grammar parameters. Those image-based impostors are generated automatically upon loading of the terminal geometries: each terminal shape is evaluated and rendered simultaneously from several views on graphics hardware (Figure 7.5). The resulting photometric attributes and normal map of the terminal are then stored for later use. Once per-terminal impostors are generated, we may replace a terminal shape symbol either by its geometric models or its impostor representation.

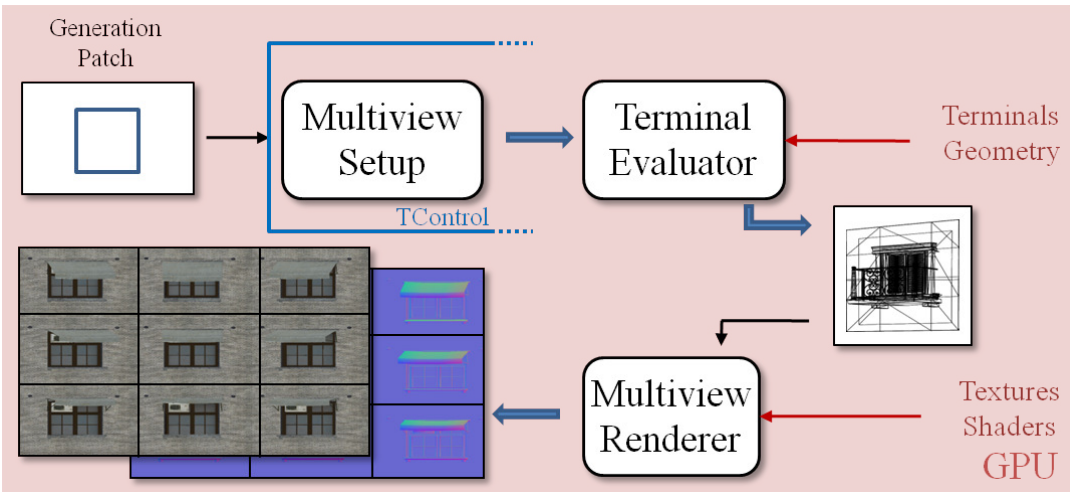


Figure 7.5: We generate impostors on the fly. For each terminal shape, we generate it using the same principle as for the terminal evaluation described in Chapter 4 (*i.e.*, with the hardware tessellator of the GPU). By specifying a multi setup, we are able to render the same geometry from various view angles in a single render pass. We store both photometric and normal information for later use at run-time.

7.3.2 LODs description

Starting from the LOD of highest quality to the lowest one, respectively LOD4 to LOD0, we explain each one according to the Figure 7.6. First, LOD4 is generated using the entire pipeline, with classical rule expansion and terminal evaluation stages. It is the only level who actually generates geometry. Then LOD3 and LOD2 only require a terminal set. Difference with LOD4 is the geometric terminal shapes are substituted by image-base impostors. For LOD3 we interpolate between the four closest multi-view impostors depending on the angle between the camera and the terminal shape preserving parallax effects. LOD2 only uses the central view impostor.

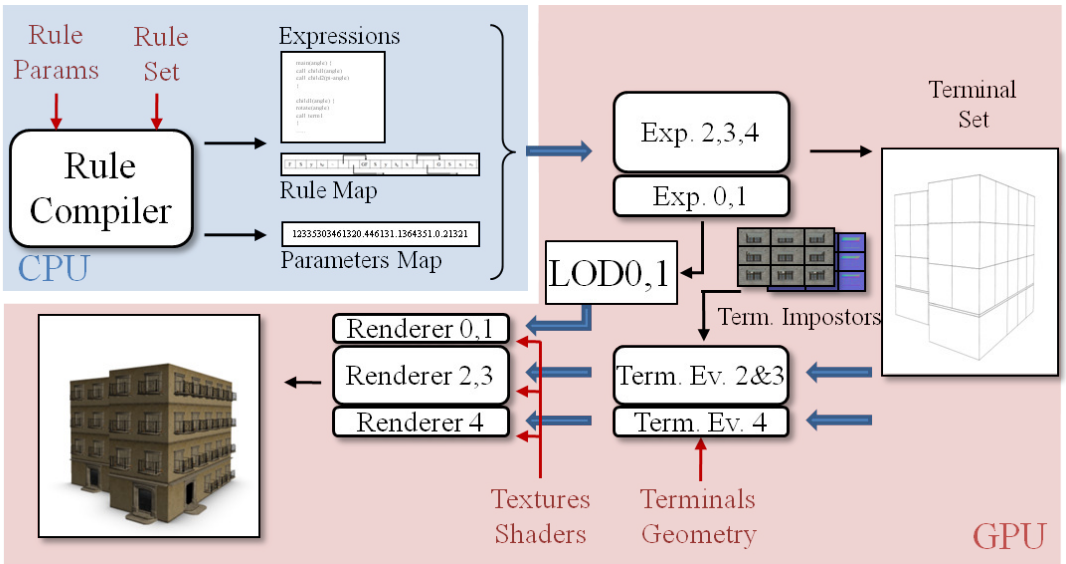


Figure 7.6: The management of levels of detail is integrated within our pipeline for interactive generation and visualization of large-scale sceneries.

Lower LODs avoid the generation of a terminal set. Instead, we generate a simplified extruded set representing the overall shape of the object. However, the high frequencies due to the presence of the terminals remain visible even from long distances. To overcome this problem, LOD1 borrows the idea of lazy per-pixel grammar expansion of [MGHS11]. The object is then simply represented by a base volume, on which the grammar is evaluated on a per-pixel basis. Depending on the terminal visible through the pixel the central view of the terminal impostor is sampled to ensure coherency with the higher levels. Finally, objects covering very few pixels on the image are replaced by a base textured volume in LOD0.

Figure 7.7 shows these five LODs applied on a building and scaled to represent the screen coverage of each one. While the choice of LODs is automatically performed at run-time based on the distance to the camera, LOD4/3/2 allow a per-terminal shapes LOD selection (Figure 7.8). Indeed as all terminal shapes are independent from each other during terminal evaluation, providing a fine-grained LOD selection.

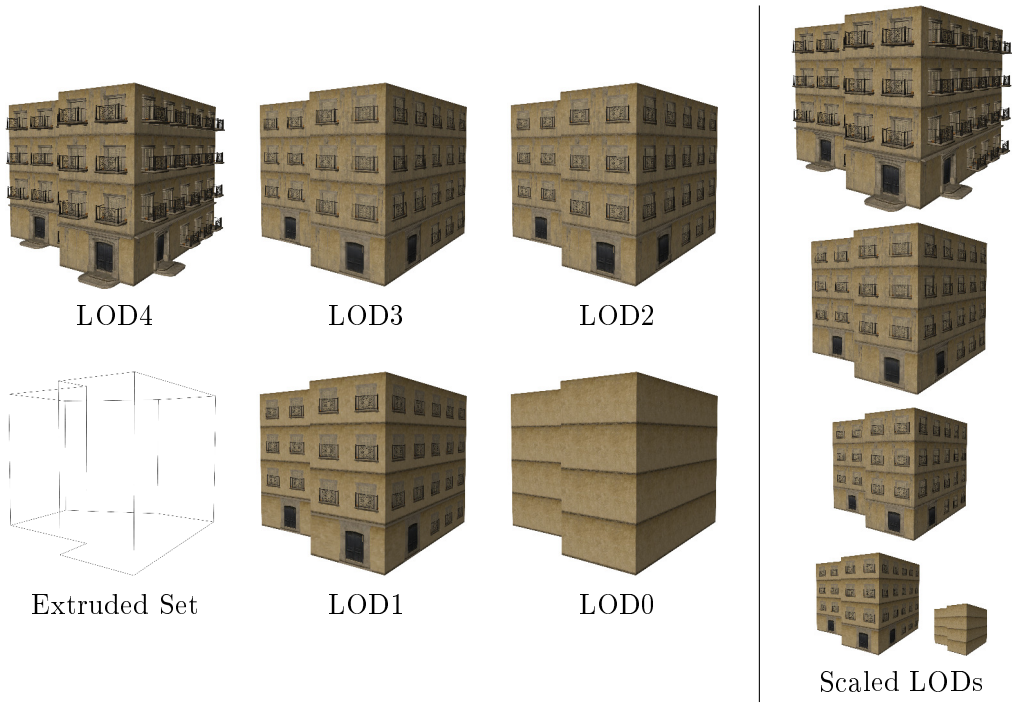
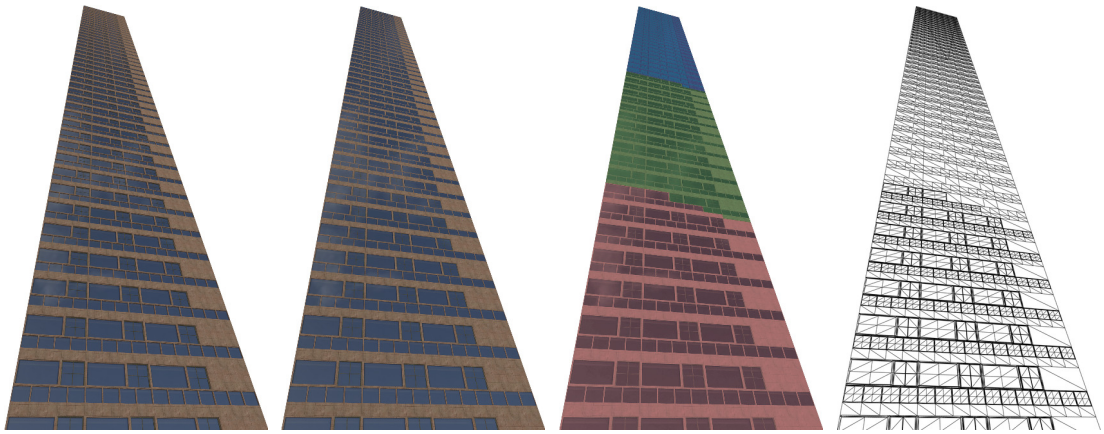


Figure 7.7: GPU Shape Grammars LOD scheme. Levels 2 to 4 are based on the full terminal set: entire terminal geometry (LOD4) or image-based impostors (LOD3/2). The LOD of each terminal is chosen independently on graphics hardware, resulting in the simultaneous use of several LODs in a single facade (right). The coarser levels rely on a simplified terminal set: LOD1 applies per-pixel grammar expansion, while farthest objects are simply textured (LOD0).



(a) Full detail (b) LOD enabled (c) LOD ranges (d) LOD shapes

Figure 7.8: Fully expanded grammar-based LOD allows for fine-grained per terminal shape LOD selection.

7.3.3 LOD transition

Discrete levels of detail usually exhibit abrupt LOD changes, introducing popping artif. We address this problem using a smooth and inexpensive LOD blending. Between LOD0 and LOD1 we simply blend the default texture with the result of per-pixel grammar expansion. The results of LOD1 and LOD2 is identical (compositing of terminal textures) and hence not prone to popping. The transition to LOD3 is then simply performed by blending of the views contained in the impostors. The last level involves the evaluation of the terminal geometries and generates the protruding features of the terminals. We avoid popping between LOD3 and LOD4 by progressively flattening the geometry as the coverage of the object decreases. This allows LOD4 to preserve a smooth transition to LOD3, while exhibiting the required highly detailed geometry in the closeup views.

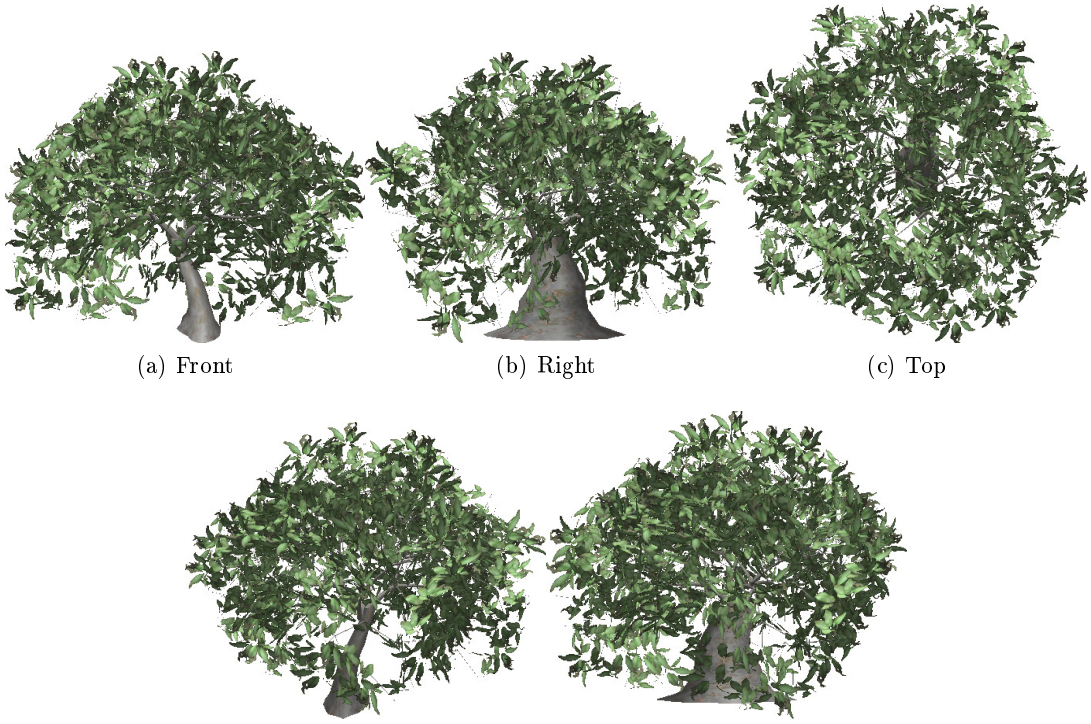
7.4 Vegetation

Vegetation shares the same idea than architecture for designing LODs. In order to relieve the geometric load of the GPU, we substitute farther trees by image-based impostors while closest ones remain actual geometry. However contrary to buildings where we defined five LODs, we propose two LODs for vegetation.

Because vegetation and architecture grammars are intrinsically different in their conception, we cannot use the same LOD technique developed for architectural models on vegetation ones. Typically, tree modeling consists in many growing rules to generate the branches and few reduction rules. Stopping the expansion rules at the first growing rule results in a unacceptable LOD, losing the global structure. In order to capture a consistent structure, we could allow a deeper expansion up to the final extrusion rules. While this method would work, it would also lead to a higher memory consumption, reducing the benefit of a lower quality LOD. Instead we adopt a different approach that use a billboard technique, where a tree is substituted by only 3 quadrilaterals (Figure 7.10). Then we texture each side of the billboard planes with object-based impostors. We developed only two LODs, the highest quality LOD being the actual generated geometry, and the lowest one being billboard-based.

7.4.1 Impostors

Instead of generating per-terminal shape impostors as in architecture case, per-object impostors are generated. We no longer shoot every terminal shapes in a hemispherical multi-view process. Impostor shooting is performed all around each generated tree using the GPU (Figure 7.9), storing into texture buffers photometric and normal information to reconstruct consistent color and lighting. The main drawbacks of per-object impostors is the need to reprocess the multiview impostor generation at parameters update because we capture the overall shape of object.



(a) Front (b) Right (c) Top

(d) Back (e) Left

Figure 7.9: Per-object impostors are generated for vegetation.

(a) Billboard (b) Textured (c) Lighted

Figure 7.10: The lower tree LOD is a texture map billboard.

7.4.2 LOD description

As said previously, we defined two LODs selected according to the camera distance. The LOD1 corresponds to the geometric level whereas the LOD0 is the image-based impostor of the generated tree. Contrary to facade elements that were rendered onto the simple quadrilateral support, we use a cross billboard impostor (Figure 7.10). Indeed, trees are objects visible all over the sphere (the hemisphere for a facade) and necessitate a 360° impostor. Each face of the billboard is assigned an ID of the object side, thus we texture the billboard by fetching the indicating view side. We also use the normal information to reconstruct a consistent lighting. Other methods exist for creating LODs for trees such as billboard clouds [DDSD03] but they needed several billboards to reconstruct parallax effects, thus increasing the geometry complexity. While our approach relies on very simple billboards, it allows to generate much more trees.

7.5 Caching, clustering and culling

LODs techniques presented for architecture in Section 7.3 and for vegetation in Section 7.4 allow to relieve the geometric load of the generated objects and to increase performances in case of thousand objects scenes. In order to reach even better performances, our pipeline also adopted advanced rendering optimization methods such as caching, clustering and culling.

7.5.1 Caching

GPU shape grammars pipeline is composed of two GPU components allowing to cache the results: the rule expander and the terminal evaluator. One may cache the terminal set generated at rule expanding, or the terminal shapes generated at terminal evaluation. Caching all results allows to deactivate the two previous stages and only performs the rendering, reaching high frame rates. However, such a caching process involves a higher memory consumption on the GPU, especially concerning the terminal evaluation stage. In order to reach interactive frame rate during navigation inside massive scenes, we found a balance between memory consumption and caching. By activating the cache of the rule expanding results, we avoid to recompute the grammar derivation every frame. Moreover, as this stage only generates a light terminal set, we optimize the graphics memory usage. We let the terminal evaluation stage be performed on-the-fly. As long as generation parameters does not change, the cache remains valid. But as soon as a parameter changes, we reactivate the rule expansion.

7.5.2 Clustering

GPU shape grammars performs parallel procedural generation on the GPU. By regrouping input 1D atoms together inside a vertex buffer, we can take advantage of this pipeline to generate multiple objects in parallel. For massive scenes, we combine batching with scene partitioning. We define clusters on the scene, and every input 1D atom belonging to the same grammar and to the same cluster (see Figure 7.11) are batched together.

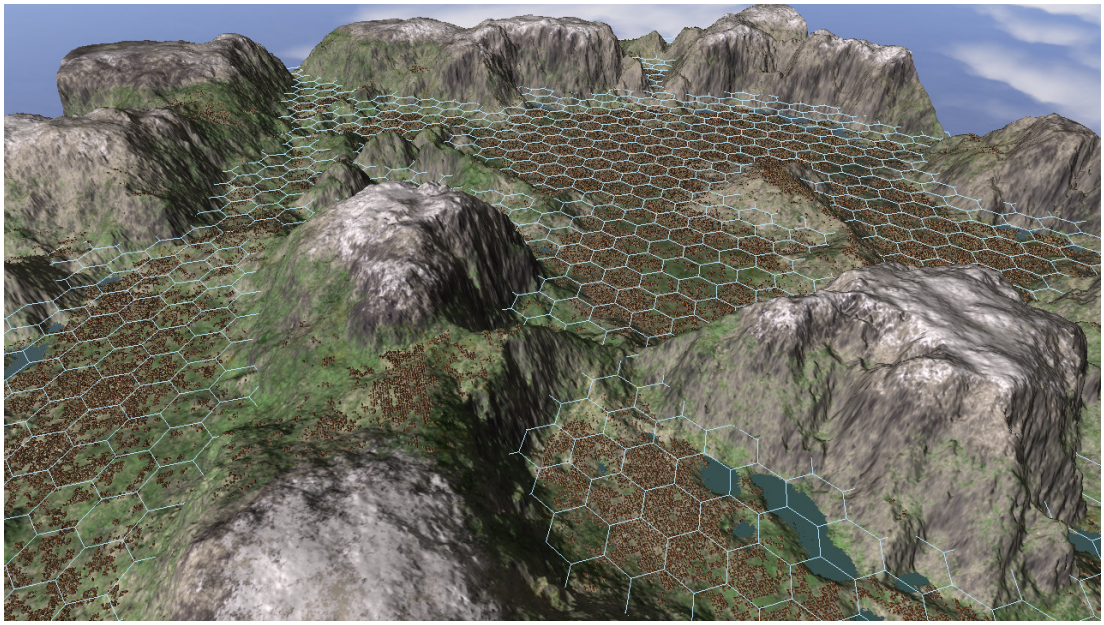


Figure 7.11: Our massive city scene is subdivided into 780 hexagonal cells corresponding to the object batches. Terminal sets are then generated on a per-cell basis.

Thus when navigating, every 1D atoms of a cluster share the same grammar-based LOD (*i.e.*, terminal set or extruded set). Finally, the second LOD is still chosen per terminal shapes. Coupling with a gross frustum culling of the clusters performed on the CPU, we are able to quickly eliminate non visible objects before any generation or rendering.

7.5.3 Fine-grain culling on GPU

Culling techniques avoid computation on non visible triangles by discarding them. As we decided to cache the results of the rule expansion on a GPU buffer, we can not use any culling method at this stage. However we perform terminal evaluation every frame, and to avoid naive geometry generation we add a GPU-based culling step. Our culling method is integrated within the terminal evaluation stage and performs two levels of culling. First, based on the terminal set we may discard some terminal quadrilaterals at tessellation control shading stage, thus before geometry generation. Terminals outside the frustum and those back-facing the camera are considered non visible and discarded. Second, during geometry generation at geometry shader stage we may discard non visible triangles. Finally, only visible triangles are generated and fragment shading stage is thus faster.

7.6 Results

We integrate our massive management system within the GPU shape grammars pipeline to the generation and tuning of buildings and vegetation in real-time. The presented images and timings were obtained at a resolution of 1280 × 720 using an Intel Xeon X5680 3.36GHz processor running a NVIDIA GeForce GTX 480 GPU.

The entire structure of the skyscrapers scene (Figure 7.12) contains 25K terminals (1.25M polygons) and is generated in 21ms. Upon visualization, the level of detail of each terminal is chosen according to the distance to the viewer, yielding a rendering speed of 21.2fps with on the fly generation, and 38.4fps by storing the terminal set (1.9MB) within graphics memory. If stored, the detailed geometry for this scene would take approximately 115MB. As our method features building-grained parallelization, 12 similar towers (100K terminals total) are generated in 26ms.

Figure 7.12: The skyscrapers comprise up to 209 floors, generated in 21ms and rendered at a minimum of 38.4fps using our massive management system within the GPU shape grammars pipeline.

Our main test scene is a massive city comprising 116573 buildings using various grammars and terminal shapes and 561280 trees of 7 different species (see Figures 7.13 to 7.16). The buildings are generated using two distinct grammars: First, the grammar for the business district buildings comprises 28 rules, 3 of which being conditional. The other buildings are generated using 40 distinct rules, including 8 stochastic rules. The buildings generated using this second grammar use 12 unique terminal shapes. The impostors for each shape are generated at a resolution of 256 × 256, yielding a total memory footprint of 25MB.

The city is divided into hexagonal cells (Figure 7.11). When LODs 2-4 are needed

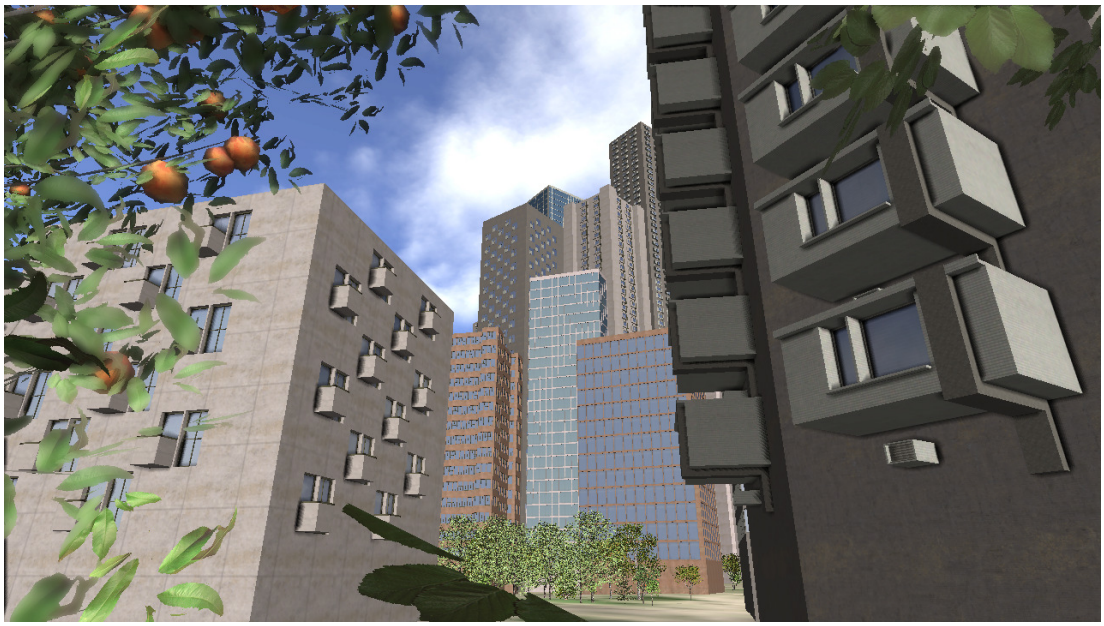


Figure 7.13: Close-up view on the business district - 8,1 fps

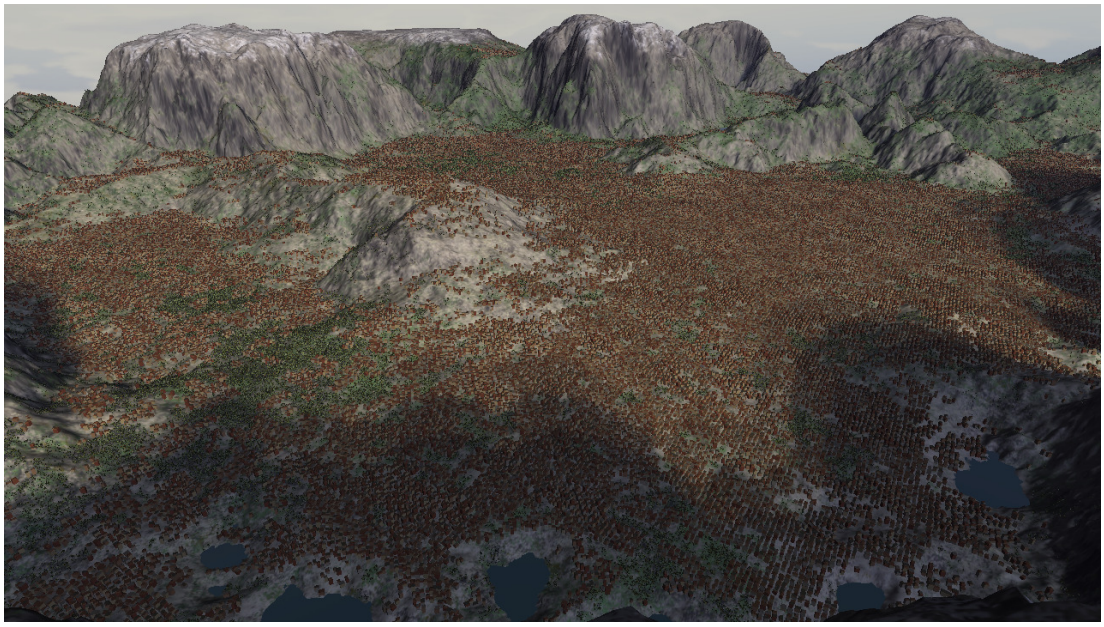


Figure 7.14: Large view of the thousands - 8,9 fps

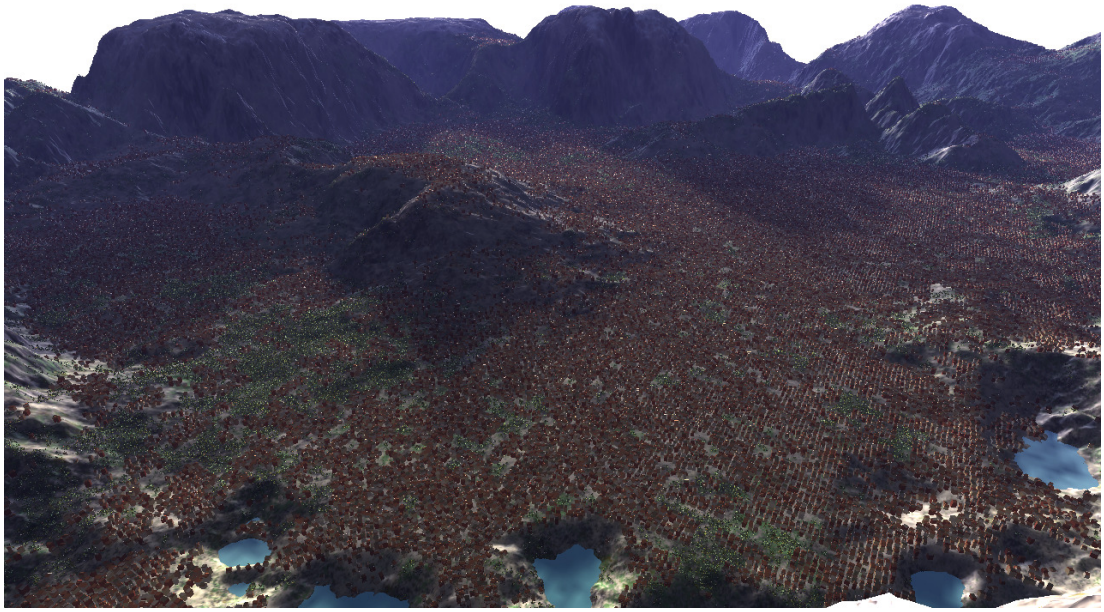


Figure 7.15: We may change lighting conditions on-the-fly - 8,7 fps

Figure 7.16: Business district and residential buildings generated from two grammars - 13,8 fps

the batch of terminal sets of a cell is generated on the fly and stored in graphics memory (5MB per cell on average). The higher levels of detail are then used within the cell, depending on the viewing distance of each terminal. The farther cells are rendered using LODs 0-1 for high performance rendering. In particular, the ray-traced grammars of LOD1 are visually equivalent to LOD2, providing a smooth transition across representations. The scene is rendered at 7-15fps with on the fly generation of the batch terminal sets for the cells.

The vegetation is generated using a grammar and LOD management. A separate billboard-based impostor is then generated for each tree type. This scene comprises 7 unique tree models, each one corresponding to one grammar and one parameters set. We decided to limit the number of tree species because we have to generate and store one multiview impostor per generated tree.

The representation of the detailed geometry for the entire city would take approximately 2.3TB, exceeding by far the memory available of high-end graphics hardware. Using GPU Shape Grammars the entire representation for the scene is reduced to an average of 900MB, pushing the size of renderable models to new limits.

Besides real-time navigation, generation parameters of any arbitrary set of buildings can be edited in place interactively. GPU shape grammars are then also a valuable tool for editing and fine tuning of complex environments. This capability finds a particular use for the production of massive assets for movie post-production. In this case, the detailed geometry can be read back from graphics hardware to generate final images using a production renderer. We could also use our system for large environment games where we would render limitless scenes on-the-fly. Such games just need to generate new set of parameters on-the-fly to create a unique environment.

7.7 Conclusion

We introduced a massive management system integrated within the GPU shape grammars pipeline allowing interactive generation, edition and visualization of huge scenes containing thousands of complex objects. While caching the rule expansion stage results on the GPU memory, we benefit from a low memory cost of the generated terminal set. By coupling with on the fly terminal evaluation and LODs, we are able to reach interactive frame rate on massive scenes. Our LOD system performs automatic impostor generation, and provides various image-based levels limiting visual artifacts. Thanks to our solution, artists may edit a single building interactively inside the overall scene, or a entire batch of buildings. It thus allows to drastically reduce the time spend by artists on environment edition contrary to state-of-art methods.

We only applied our architectural LOD system for facades and not for the roofs. In such case, we could specify into the grammar multiple extrude sets (*i.e.*, one for the facade, another for the roof section) and use different base texture for the lower LOD. For the impostor-based LOD, we could use tiles impostor. However, covering roof sections with unitary tiles will give a high number of terminal shapes and lead to a higher memory consumption. Using batch of tiles could limit this problem by generating

less terminals. Moreover, in case of mansard windows being both part of the facade and the roof, we could use another extrude set corresponding to its lower LOD.

Chapter 8

Conclusion

This thesis presented a parallel approach for procedural generation. To conclude, we summarize our contributions. Finally, we discuss perspectives for future work.

8.1 Contributions

We introduced a parallel procedural pipeline, bringing the massive power of parallel hardware to the procedural generation and rendering of highly detailed models. Context-based extensions are introduced to build consistent models according to internal and external contexts. Our approach also integrates a scalable framework for interactive generation and rendering of massive environments.

8.1.1 Parallel procedural generation based on independent 1D atoms

We first designed a parallel procedural pipeline taking advantage of the massively parallel architectures of recent graphics hardware. As such architectures efficiently handle many instances of simple data, we introduced simple 1D atoms and a segment-based expansion to address the GPUs. Initial input objects are subdivided at the segment level: the 1D atoms. Then segment-based expansion is performed in parallel for each 1D atom on the GPU. Our pipeline is decomposed into three stages: the rule compiler, the rule expander and the terminal evaluator. First, the CPU rule compiler converts the rule set and parameters into a GPU-interpretable rule map and shader expressions. Then, for each 1D atom in parallel on the GPU, the rule expander traverses the rule map according to the segment-based expansion and evaluates the expressions on-the-fly, generating the terminal sets: the lightweight structures of the objects. Finally, the GPU-based terminal evaluator instantiates the terminal geometries at render time. Our framework also considers the hardware tessellator to efficiently generate the geometry

with both the rule expander and the terminal evaluator. We assessed our pipeline with architecture and vegetation models interactively generated and rendered.

8.1.2 Internal context parallelization

With the 1D atoms decomposition, the internal contexts of input objects are lost. As 1D atoms belonging to the same input model are independent from each other, internally consistent models cannot be generated. Our solution specifies local contexts for the 1D atoms from the internal contexts. Then, *Join* and *Project* rules are introduced to create consistent structure from local contexts. These 1D atom local contexts are used to guide consistent generation over input objects. For instance, compelling complex grammar-based roofs are generated at interactive time using our system.

8.1.3 External context-sensitive grammars

Controlling grammar rules behavior from contexts external to the input object can help the artist to edit grammar-based models. We thus introduced texture and surface contexts to constrain the grammar rules. Using a texture lookup accessor, any rule can query either pre-generated textures or on-the-fly paintings. Texture contexts are then used as grammar constraints. In addition, we provide surface contexts to allow consistent growth expansion on underlying surfaces. Represented as indirection geometry image, the surface context is processed by the *Marching* rule. Integrated within our parallel procedural pipeline, our approach allows for interactive generation of growth models such as ivy plants onto complex surfaces.

8.1.4 Parallel grammars scalability

Our parallel procedural pipeline is designed to interactively generate and render hundreds of objects. To handle very-large environment composed of thousands of elements, related memory consumption and geometric complexities issues are addressed with scalable grammars. To do so, a complete LOD system based on advanced rendering optimization strategies are integrated within our pipeline. Our LOD system is designed to seamlessly substitute geometry by texture-based impostors, preventing from visual artifacts. Associated with culling, caching and clustering methods, huge scenes containing thousands of objects are generated and rendered at interactive time.

8.2 Perspectives

Our pipeline allows for interactive generation and rendering of massive environments never before seen. While it benefits from the massive parallel power of the GPUs, the two GPU-based stages (rule expander and terminal evaluator) also take advantage of the hardware tessellator. However, such graphics feature is not available in many hardware of the production industry, preventing an integration of our system within existing rendering pipelines. This could be addressed using a more traditional feature:

the geometry instancing. This feature allows to render many instances of an object with a single draw call using matrix transformations stored into GPU buffers. In order to substitute the hardware tessellator for geometry instancing, the rule expander should generate matrix transformations for each terminal shapes into a GPU buffer. Synchronization of concurrent threads to write the buffer should be addressed very carefully to preserve high parallelism performances. Then the terminal evaluator stage would perform series of instancing draw calls, for each terminal shape. Such method implies more draw calls per batch and thus a higher CPU cost. On the counterpart it also avoids the cost of our redirected usage of the tessellator.

Regarding mobile devices, one can notice that even latest OpenGL specification (OpenGL|ES 3.0), does not provide hardware tessellation. However, transform feedbacks for storing intermediate results are now available. Depending on the success of the geometry instancing approach instead of the tessellator, our method could also address mobile devices were efficient generation and small storage are necessary. In addition, as grammar rule sets encode highly-detailed objects very compactly, such grammar rules could be quickly streamed over low-bandwidth networks. Thanks to this, mobile clients could achieve interactive generation and rendering using our pipeline.

As seen in Chapter 4, the rule expansion is evaluated in a single pass within the GPU. However, depending on the application, multiple expansion passes could be useful to increase the parallelism performances. For instance, an artist could start from a terrain model and generate seeds on-the-fly attached to various grammars, according to a texture context. Then seeds would be processed in second expansion pass in parallel per grammar used. To do this, an idea is to extend the grammar language to provide in/out buffer specifications and to build the parallel procedural pipeline with multi-pass rule expansions accordingly.

Finally, growth on surfaces are possible thanks to the *Marching* rule introduced, using our surface context representation: indirection geometry image. However, such representation is based on pre-generated models, preventing growth on grammar-based models. In this case, the surface context would be the grammar itself. A solution would be to perform a lazy ray-tracing through the grammar [MGHS11] to find generated surfaces.

Chapter 9

Résumé en Français

9.1 Introduction

Afin de créer des productions toujours plus réalistes, les industries du jeu vidéo et du cinéma cherchent à générer des environnements de plus en plus larges et complexes. Cependant, la modélisation des objets 3D dans de tels décors se révèle très couteuse, que ce soit en temps de génération, en temps d'affichage, ou encore en quantité de stockage. Tous ces inconvénients empêchent l'édition et la pré-visualisation interactive de la scène résultante.

L'objectif de cette thèse est de permettre la génération et l'affichage en temps interactif des objets peuplant ces scènes massives, tels que la végétation ou l'architecture. Pour y répondre, nous adoptons une approche basée sur le parallélisme et sur les méthodes de génération procédurale pour exploiter efficacement la puissance des cartes graphiques (GPUs) modernes.

D'un côté, les méthodes de génération procédurale permettent de créer facilement une grande variété d'objets, en exploitant les similarités dans une même classe d'objets comme les plantes et les bâtiments. Plus particulièrement, la modélisation par règles de grammaire offre un outil de haut niveau pour décrire ces objets de façon très compressée, et permet d'obtenir une infinité de résultats similaires. Grâce à leur rôle d'amplification, ces méthodes sont utilisées pour modéliser les environnements dans certains films tels que *King Kong* [Whi06] ou *Man of Steel* [Esr12].

D'un autre côté, la génération procédurale est effectuée généralement sur le CPU puis le résultat est transféré sur le GPU pour effectuer l'affichage. Ce schéma de pré-amplification résulte en un problème de consommation mémoire et de coût de transfert sur le bus CPU/GPU, empêchant une édition interactive dans le cas d'environments complexes. C'est pourquoi nous cherchons à réaliser l'étape d'amplification directement sur le GPU. Grâce à l'architecture des cartes graphiques récentes qui permet un parallélisme de données performant, nous pouvons alors atteindre des performances interactives lors de la génération de scènes massives.

9.1.1 Contributions

Nous proposons un système permettant la génération procédurale en parallèle sur le GPU en temps interactif, basé sur 4 contributions :

- Nous adoptons une approche d'expansion indépendante par segment, permettant une amplification des données en parallèle.
- Nous étendons ce système pour générer des modèles basés sur un contexte structurel interne tels que les toits.
- Nous présentons aussi une solution utilisant des contextes externes pour contrôler facilement les grammaires par le biais de texture et permettant une expansion sur des surfaces.
- Enfin nous intégrons un système de niveaux de détail et des méthodes d'optimisation permettant la génération, l'édition et la visualisation interactives d'environnements à grande échelle.

Grâce à notre système il est possible de générer et d'afficher des scènes comprenant des milliers de bâtiments et d'arbres interactivement, ce qui représenterait environ 2 tera-octets de données avec des méthodes classiques, soit environ 2000 fois la taille mémoire disponible sur une carte graphique standard.

9.2 État de l'art sur la génération procédurale

Les méthodes procédurales permettent de modéliser différents types d'objets, tels que les terrains [SDKT+09, GGG+13, HGA+10], les réseaux routiers [GG10, GPMG10], les environnements urbains [PM01, CEW+08, VKW+12], l'architecture [WWSR03, MWH+06, WOD09], les intérieurs [MSK10, LHP11] ou encore les textures [EMP+02, All13]. Parmi les méthodes procédurales, les grammaires représentent la structure d'objets par une succession de règles de description. Ainsi les plantes, les bâtiments ou encore les terrains peuvent être très facilement modélisés de cette manière. Par exemple, un arbre résulte d'un enchainement de règles de croissance. Dans le cas d'un bâtiment, chaque façade est généralement décomposée en un rez-de-chaussée et plusieurs étages, lesquels sont aussi subdivisés en fenêtre, portes, balcons, etc.

Parmi les méthodes basées sur des grammaires, les *L-systèmes* sont dédiés aux opérations de croissance et permettent de modéliser de la végétation par le biais de règles de réécriture [Lin68]. Ces systèmes ont aussi été étendus pour modéliser des bâtiments [PM01, MPB05]. Cependant, d'autres grammaires issues des *shape grammars* [SG71] se sont révélées beaucoup plus adaptées dans le cadre de l'architecture, comme par exemple les *split grammars* [WWSR03], et les *CGA shape grammars* [MWH+06]. Dans ces différentes approches, les règles de grammaire décomposent une géométrie initiale en plusieurs sous parties récursivement, jusqu'à obtenir des éléments terminaux (étape de dérivation/développement). En associant des géométries détaillées aux éléments terminaux (étape d'interprétation/évaluation), il est alors possible d'obtenir le modèle

final. Toutes ces méthodes permettent de compresser un modèle détaillé sous forme de grammaire très légère. Cependant elles nécessitent un temps de génération empêchant toute interactivité pour l'utilisateur. En effet, dans les pipelines de génération classique (Figure 9.1), la géométrie est amplifiée en utilisant le CPU et peut éventuellement être stockée sur disque dur. Elle est ensuite transférée au GPU pour procéder à l'affichage. Ce transfert est très couteux et représente un véritable goulot d'étranglement.

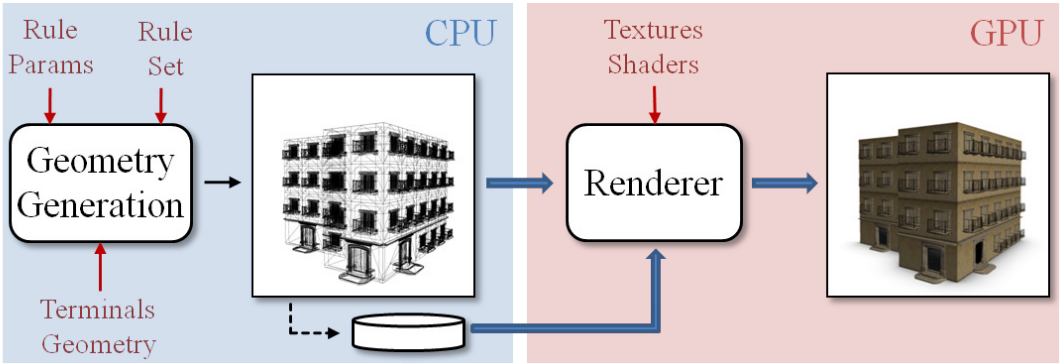


Figure 9.1: Dans un pipeline de génération procédurale classique.

Dans l'optique d'éviter notamment ces coûts de transfert, des approches récentes génèrent directement les modèles amplifiés sur la carte graphique en temps interactif. Les *L-systèmes* peuvent ainsi être générés en parallèle par calcul générique [LWW10]. En ce qui concerne les *shape grammars*, la génération des modèles par pixel de l'image finale [HWA+10, KK11b, MGHS11] permet une édition interactive pour des scènes de centaines d'objets. Cependant ces dernières méthodes sont contraintes à un coût par pixel. Pour répondre à la problématique de génération interactive de scènes massives, nous orientons également nos recherches sur une approche utilisant efficacement le GPU. La partie suivante décrit notre première contribution, un pipeline de génération procédurale parallèle sur GPU.

9.3 Génération procédurale en parallèle basée sur des atomes 1D indépendants

Dans cette partie nous introduisons une nouvelle méthode permettant la modélisation et la visualisation interactives d'objets procéduraux complexes. Afin de combiner parallélisme et génération procédurale, nous proposons un développement de grammaire basé sur des segments, permettant à la fois des opérations de croissance et de réduction, tout en travaillant en parallèle sur des atomes 1D indépendants. Nous définissons un pipeline de génération et d'affichage de modèles procéduraux évitant le stockage des modèles détaillés (Figure 9.2). Parmi les avantages de notre solution, nous pouvons également noter la compatibilité avec les moteurs de rendu temps réel existants.

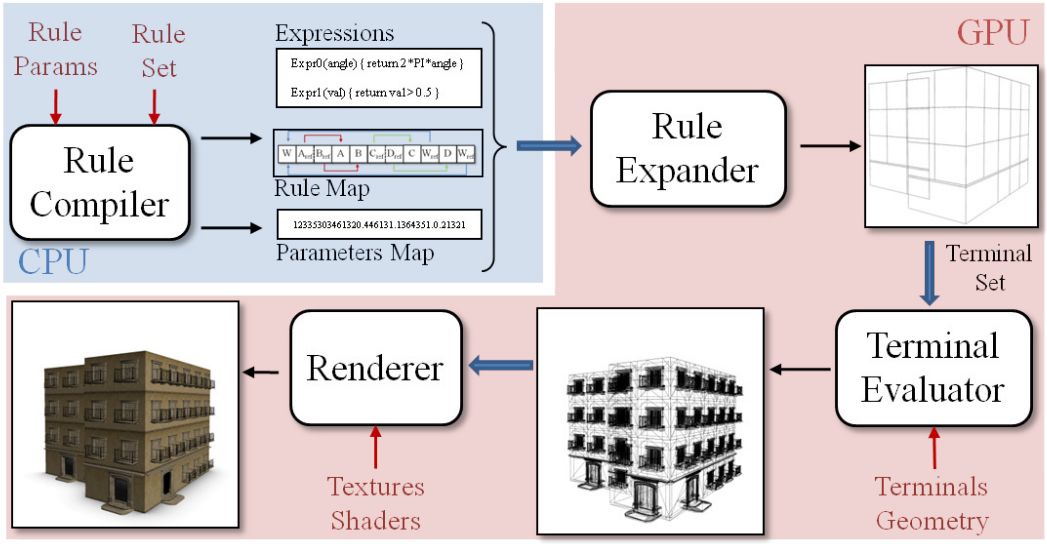


Figure 9.2: Notre pipeline permet la génération de modèles procéduraux sur GPU. Les grammaires sont compilées sur CPU en une structure efficace pour un développement de grammaire sur GPU. L'expansion de la grammaire génère une représentation intermédiaire très légère de la structure de l'objet. La géométrie détaillée est finalement générée à la volée dans une seconde étape GPU.

9.3.1 Description du pipeline

Le compilateur de règles, sur CPU, extrait de la grammaire un graphe générique ainsi que les expressions sous forme de code *shader*. Ce graphe représentant les liens de parenté entre les différentes règles est encodé sous forme de texture: la carte des règles. De même, les paramètres sont aussi encodés sous forme de texture.

Toutes ces structures sont ensuite utilisées pour le développement des règles sur GPU. Dans cette seconde étape, nous parcourons la carte des règles selon les expressions et paramètres associés pour chaque graine d'entrée afin de générer l'ensemble des terminaux décrivant la structure légère de l'objet. Les graines d'entrée sont des atomes 1D correspondant à la décomposition des données d'entrée complexes en éléments simples et indépendants, pour préserver un traitement efficace sur GPU. En conséquence, nous redéfinissons le développement de la grammaire basé sur des segments. La figure 9.3 illustre un développement.

Finalement l'étape d'association des géométries détaillées est réalisée à la volée sur GPU, évitant le stockage du résultat. Chaque élément terminal est substitué par sa géométrie, stockée dans la mémoire sur GPU. Les deux étapes sur GPU utilisent efficacement les unités de tessellation dédiées à la génération à la demande de polygones. L'étape d'affichage est compatible avec les méthodes d'affichage classiques, assurant un rendu cohérent entre les objects procéduraux et le reste de la scène.

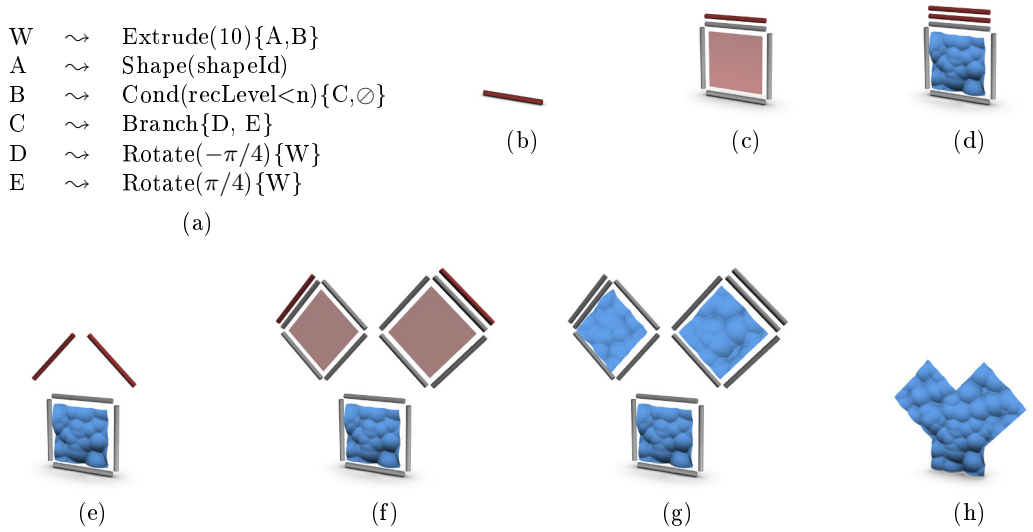


Figure 9.3: Décomposition étape par étape du développement de la grammaire (a). L'axiome est appliqué sur l'atome 1D (b), résultant en une face composée de 4 atomes 1D et d'un élément 2D (*i.e.*, un élément d'expansion), et en un atome 1D déplacé (c). L'élément d'expansion est remplacé par une géométrie terminale, alors que l'atome 1D déplacé est dupliqué en 2 branches (d), chacune suivant une rotation (e). Par récursion, les branches sont de nouveau extrudées (f) et l'on instancie des géometries terminales (g), donnant le modèle final (h).

9.3.2 Applications et résultats

Grâce à la reformulation supportant à la fois les opérations de croissance et de réduction, nous générons et éditons en temps réel des arbres et des bâtiments selon notre approche (Figure 9.4). Dans le cas des bâtiments, nous avons décomposé les empreintes au sol initiales en atomes 1D indépendants. Ainsi nous pouvons paralléliser sur le nombre de façade en entrée. Pour une même grammaire, il possible d'obtenir une grande diversité avec des géométries terminales et des jeux de paramètres différents. Pour les arbres, nous partons simplement d'un atome 1D agissant comme la graine de départ.

Nous pouvons aussi regrouper toutes les graines suivant une même grammaire ensemble car dans ce cas seuls les paramètres changent entre différents objets. Ce regroupement nous permet ainsi de profiter au maximum du parallélisme. En effet, toutes ces graines sont décomposées en atomes 1D et traitées en parallèle par la carte graphique. En regroupant les graines nous observons des paliers de temps de génération (Figure 9.5). Ces paliers montrent que plusieurs objets sont générés au coût d'un seul, et démontrent que notre approche préserve bien le parallélisme de données.

Ce pipeline générique permet donc de générer et éditer des objets procéduraux

complexes en temps interactif, en tirant parti du parallélisme et minimisant l'empreinte mémoire. Cependant la décomposition des graines en atomes 1D entraine la perte du contexte interne de la graine initiale. Une fois la graine décomposée en atomes, chaque atome se retrouve totalement indépendant, ce qui empêche la génération de modèle nécessitant un contexte interne.

(a) (b) (c)

Figure 9.4: Le développement des grammaires des bâtiments (a) et (c) prend 0.2 ms, tandis que l'évaluation des terminaux et l'affichage sont réalisés en 8.4 ms (a) et 4.0 ms (c). L'arbre (b), dont la grammaire est disponible à la section 4.6.2 est lui développé en 40 ms avec 5 récursions et évalué en 2 ms.

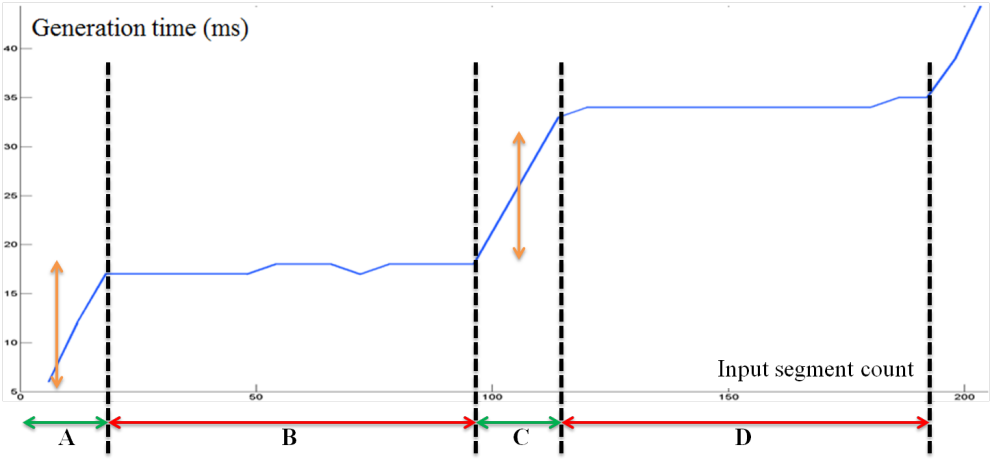


Figure 9.5: Nous observons différents paliers de temps de génération dépendant du nombre d'atomes 1D. Environ 100 atomes sont traités en parallèle à chaque tour de calcul, à un coût moindre. Par exemple, ici 90 segments sont traités au coût de 20, montrant l'efficacité du parallélisme.

9.4 Parallélisation du contexte interne

Les atomes d'entrée étant indépendant les uns des autres, ils ne peuvent pas réaliser des opérations cohérentes avec les atomes de la même graine initiale. Par exemple, chaque atome peut générer sa façade verticalement, mais ne peut pas créer les différents pans de toit. En effet, pour réaliser ce type d'opération, les atomes doivent avoir une certaine connaissance du contexte interne à la graine. Dans le cadre d'un toit, le contexte interne peut représenter les lignes de fait et les pignons. Ces informations indiquent comment assurer la cohérence du modèle.

9.4.1 Méthode

Dans cette seconde contribution nous nous intéressons à la génération de modèles nécessitant un contexte interne, en préservant le parallélisme acquis. Notre approche consiste à décomposer le contexte interne global à la graine en contextes internes locaux affectés aux différents atomes 1D. Ainsi chaque atome pourra construire sa partie de modèle de façon cohérente. Pour cela nous avons étendu le pipeline précédent (Figure 9.6).

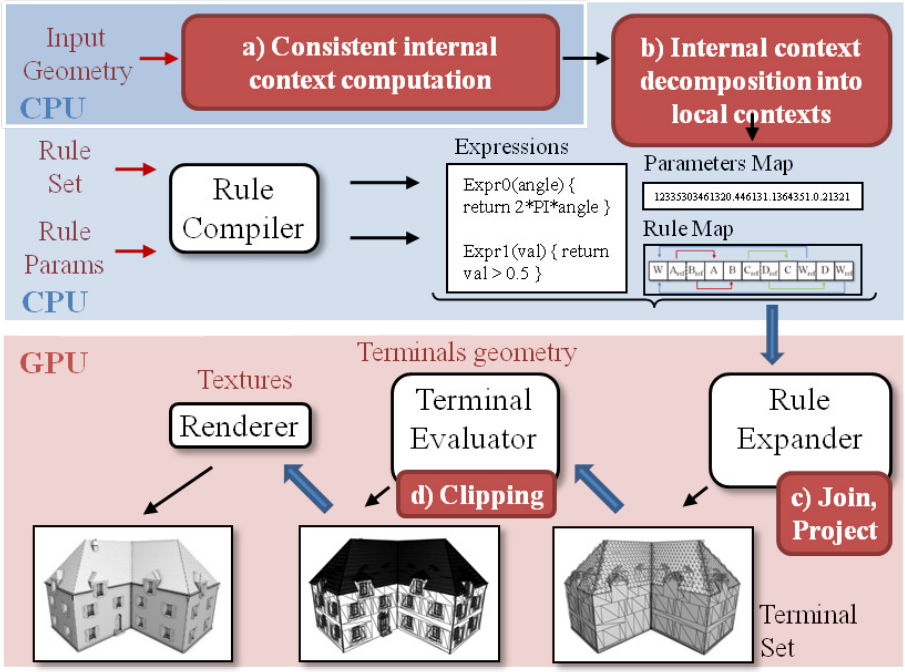


Figure 9.6: Pour générer des modèles cohérents sur une graine initiale, nous calculons d'abord un contexte interne global en pré-traitement sur CPU (a). Ce contexte interne global est converti en multiples contextes internes locaux affectés à chaque atome 1D (b). Cette information de contexte interne local est ensuite traitée par les règles *Join* et *Project* pour générer des structures cohérentes (c). Finalement les éléments terminaux sont découpés pour assurer une bonne continuité entre les atomes adjacents (d).

9.4.2 Application à la génération de toits & résultats

La décomposition du contexte interne global en contextes internes locaux définis pour chaque atome 1D nous permet de modéliser par exemple différents types de toit (Figure 9.7). Alors que nous obtenons un très haut niveau de détails en utilisant des tuiles géométriques détaillées comme élément terminal, la cohérence entre les atomes adjacents est bien assurée grâce à la découpe précise des géométries terminales. De plus, avec le contexte interne nous pouvons ajouter des éléments importants tels que des cheminées, des velux, des balcons, ou encore des fenêtres mansardées. Le même contexte interne peut être également utilisé pour modéliser des gouttières autour des toits.

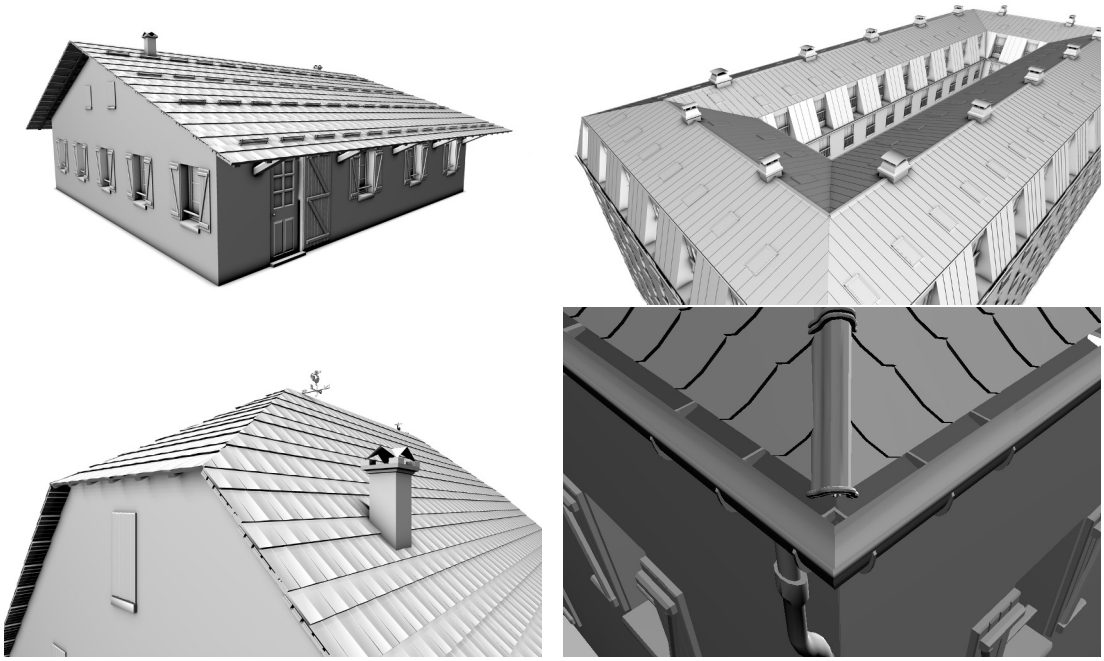


Figure 9.7: Différents types de toit très détaillés peuvent être modélisés selon notre approche.

Notre approche préserve le parallélisme et l'interactivité issus de la première contribution car chaque atome reste totalement indépendant des autres. La dépendance entre atomes étant gérée de façon similaire par la décomposition du contexte interne. Par exemple, de 1 à 20 maisons de type normande (Figure 9.7c) sont générées à un temps constant de 11.5ms (80 atomes), tandis qu'un second palier est observé de 21 à 40 maisons (160 atomes) à 22.3ms.

Grâce à cette approche, nous pouvons donc générer des modèles nécessitant un contexte interne, en parallèle sur le GPU. Cependant, nous ne pouvons toujours pas utiliser de contexte externe pour contraindre une expansion de grammaire.

9.5 Grammaires sensibles au contexte externe

Dans les approches précédentes, chaque graine était positionnée dans l'environnement, puis le modèle était généré sans aucune contrainte sur cet environnement. Dans cette troisième contribution nous souhaitons pouvoir utiliser l'environnement extérieur, c'est-à-dire le contexte externe, comme information contrôlant la grammaire. Par exemple, les caractéristiques d'un terrain (pente, composition chimique du sol, etc) peuvent influer sur les modèles d'arbres générés sur ce même terrain. De plus, nous voulons aussi permettre une expansion de grammaire sur une surface sous-jacente.

9.5.1 Méthode

Tout d'abord nous choisissons d'encoder les contextes externes sous forme de texture car c'est un format adapté pour le GPU. En ajoutant un simple accesseur texture dans la grammaire, nous pouvons très facilement interroger un contexte externe en paramètre d'une règle. Il est ainsi possible, par exemple, de contrôler une grammaire de génération d'arbre avec une carte de population indiquant les zones adéquates à chaque espèce.

Dans le cadre spécifique d'une expansion sur une surface, la grammaire a besoin de savoir tout au long de son développement où elle se situe par rapport à cette surface sous-jacente. De la même manière les contextes de surface sont encodés sont forme de texture, et plus particulièrement de *geometry image* [GGH02, SWG+03]. Avec cette représentation un maillage est décomposé en plusieurs îlots qui sont ensuite projetés séparément dans un atlas de texture. Bien que chaque texel non vide de l'atlas corresponde à un échantillon du maillage d'origine, les différents îlots sont entièrement indépendants. Pour réaliser un *marching* sur la surface en utilisant une *geometry image*, nous avons donc besoin de pointeurs d'indirection aux bordures de chacun des îlots. Ces pointeurs nous serviront par la suite à passer d'un îlot à un autre pendant le *marching*.

Nous définissons une nouvelle règle *Marching* dans la grammaire qui réalise une expansion sur une surface. Basée sur une *geometry image*, l'algorithme de marching commence à la coordonnée de texture de la graine initiale. Ensuite, selon une direction donnée, nous allons récupérer le pixel destination dont la valeur sera la position de destination de l'expansion. Au final, 4 cas peuvent être rencontrés en fonction du pixel destination (Figure 9.8):

(A) soit il appartient au même îlot que le pixel source, pas de redirection

(B) soit il correspond à un pixel d'indirection, la redirection vers l'îlot adjacent est automatique suivant la nouvelle coordonnée de texture

(C) soit il appartient à un autre îlot, nous réalisons un *marching* en arrière pour retrouver le bon pointeur d'indirection

(D) soit il appartient à un autre îlot, idem (C)

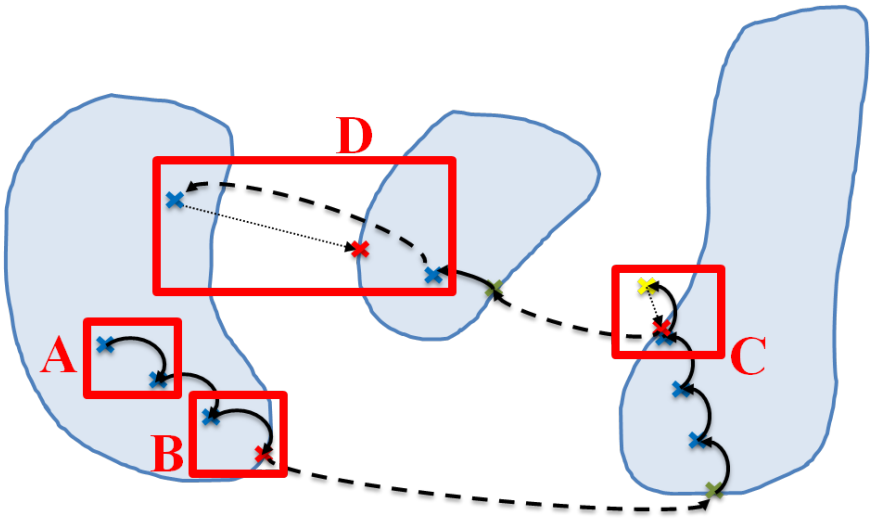


Figure 9.8: L'algorithme de *marching* permet de passer d'un îlot de la *geometry image* à une autre.

Finalement cette méthode permet de réaliser des expansions sur une surface. Nous pouvons ensuite appliquer des géométries sur les éléments terminaux. Pour éviter les discontinuités entre 2 géométries adjacentes, nous ajoutons une étape de lissage à l'instanciation des géométries terminales. Ce lissage est calculé selon des courbes de Bézier cubiques assurant une continuité de tangente (Figure 9.9).

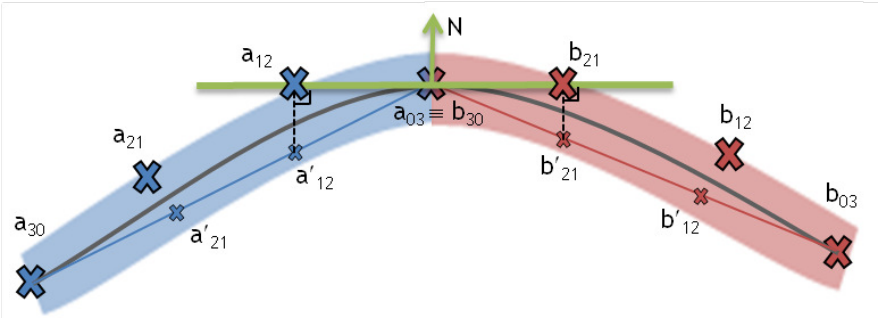


Figure 9.9: L'utilisation de courbe de Bezier cubique permet une continuité lisse entre 2 géométries terminales.

9.5.2 Application à la croissance sur des surfaces génériques & résultats

Il est possible de modéliser des croissances sur des surfaces en temps interactif en utilisant la méthode de *marching*, comme par exemple du lierre poussant sur des ruines

(Figure 9.10a). Nous pouvons aussi combiner une grammaire de croissance avec des contraintes génériques appliquées à la surface par le biais de texture. La figure 9.10b montre des zones de croissance autorisées qui sont décidées par l'utilisateur. Il est alors très simple de contrôler un développement de grammaire de cette façon en appliquant la peinture directement sur le maillage, et voir la grammaire s'adapter à ce nouveau contexte en temps interactif.

(a) Croissance libre (b) Contrainte contrainte

Figure 9.10: Le modèle de ruines est couvert par 18 lierres développés en parallèle (a). Des centaines de milliers d'éléments terminaux sont générés en 421 ms, et instanciés en 5.17 millions de polygones en 87 ms. L'utilisateur peut également peindre des contraintes de croissance sur le modèle de support et voir la génération s'adapter en temps interactif (b).

Le pipeline tel que présenté et étendu dans les parties précédentes est capable de générer des centaines d'objets en temps interactif mais on observe une chute des performances pour des milliers d'objets.

9.6 Mise à l'échelle des grammaires parallèles

Dans cette quatrième contribution nous allons étendre ce pipeline pour pouvoir passer à l'échelle, et ainsi générer des environnements massifs avec des centaines de milliers d'objets en temps interactif.

9.6.1 Méthode

Tout d'abord nous procédons à une subdivision de la scène en cellules. Pour chacune des cellules, toutes les graines associées à une même grammaire sont regroupées ensemble

pour profiter du parallélisme. Ensuite un niveau de détail géométrique est choisi selon la distance à la caméra pour chaque cellule, ce qui nous permet d'alléger le coût mémoire de la scène. Dans le cadre de l'architecture, le niveau bas correspond à une simple extrusion des bâtiments, alors que le niveau haut comprend le développement entier de la grammaire avec toutes les opérations de réductions. En complément à ce niveau de détail géométrique, 5 niveaux sont définis en utilisant des imposteurs multi-vue par géométrie terminale. L'idée est de remplacer la complexité géométrique des terminaux par une simple texture. Ainsi au fur et à mesure du rapprochement de la caméra, la qualité de rendu est privilégiée au détriment de l'espace mémoire. Comme à chaque instant seules les cellules les plus proches sont à un haut niveau de détail, cette approche nous permet d'équilibrer la charge sur des scènes massives, tout en conservant une édition interactive.

9.6.2 Application aux environnements massifs & résultats

L'extension du pipeline permet de générer des scènes massives comprenant des centaines de milliers d'objets en temps interactif. La scène en figure 9.11 comprend 116573 bâtiments et 561280 arbres, avec des grammaires différentes. Cette scène est rendue interactivement entre 7 et 15 fps en choisissant et générant à la volée les niveaux de détail adéquats. Grâce à l'utilisation de niveaux de détail et d'imposteurs à la place des terminaux géométriques, cette scène massive ne représente que 900 Mo en mémoire GPU au lieu de 2.3 To.

Figure 9.11: Scène massive rendue en temps interactif.

9.7 Conclusion

Dans cette thèse nous avons présenté notre approche parallèle pour faire de la génération procédurale. Nous allons tout d'abord résumer les contributions de cette thèse, puis nous verrons quelques perspectives de travaux futurs.

9.7.1 Contributions

Premièrement nous avons créé un pipeline de génération procédurale parallèle tirant bénéfice des cartes graphiques récentes. Comme les GPUs exécutent efficacement des

calculs simples sur un grand nombre de données indépendantes en parallèle, nous avons choisi de décomposer les entrées initiales en atomes 1D indépendants, pour lesquels un développement de grammaire simple, basé sur des segments, est exécuté sur le GPU. Ce pipeline est composé de 3 parties: un compilateur de règles, un développeur de règles et un instanciateur de terminaux. La compilation des règles s'effectue sur CPU et convertit l'ensemble des règles et les paramètres sous forme de structures adaptées pour le GPU. Ensuite pour toutes les graines d'entrée, la grammaire est développée en parallèle sur GPU et génère l'ensemble des terminaux. Cet ensemble correspond à la structure intermédiaire de l'objet. Finalement, l'évaluateur de terminaux instancie sur GPU les géométries à la place des terminaux. Grâce à cette formulation du développement de la grammaire, nous pouvons efficacement créer aussi bien des bâtiments que des plantes, en temps interactif.

Nous avons étendu ce pipeline pour pouvoir modéliser des objets nécessitant un contexte interne pour assurer avec une cohérence globale malgré la décomposition en atomes indépendants. Pour ce faire nous utilisons également une décomposition du contexte interne global, calculé sur la graine initiale, en contextes internes locaux affectés à chaque atome 1D. Ces contextes locaux sont ensuite traités lors du développement de la grammaire. Par exemple nous pouvons utiliser cette approche pour modéliser des toits très détaillés.

Dans une troisième contribution nous avons aussi pris en compte les contextes externes aux objets, c'est-à-dire les informations de l'environnement. En encodant ces contextes sous forme de textures, il est aisé de contraindre un développement de grammaire selon un contexte externe. De plus, nous avons défini une règle de *marching* pour générer des expansions sur des surfaces. Notre approche permet par exemple de modéliser des croissances de lierre sur des surfaces complexes, tout en préservant une édition des contextes en temps interactif en peignant directement sur la surface.

Finalement nous avons proposé une solution pour permettre la génération et l'affichage interactifs de scènes massives composées de milliers d'objets. Notre approche réside en un équilibrage entre la qualité de rendu et le coût mémoire. Nous avons étendu le pipeline avec des méthodes d'optimisation de scènes et de niveaux de détail. Il est alors possible de créer des scènes massives de centaines de milliers d'objets. L'intégration de ces différentes méthodes nous permet de naviguer interactivement dans de telles scènes, tout en préservant une édition à la volée d'un ou plusieurs objets.

9.7.2 Perspectives

Notre pipeline permet une génération et un affichage interactifs d'environnements massifs encore jamais vu. Notre approche bénéficie du parallélisme des cartes graphiques récentes mais aussi de la *tessellation* matérielle. Cependant cette fonctionnalité n'est pas disponible dans une majeure partie des cartes graphiques dans l'industrie de la production cinématographique. Pour intégrer notre pipeline en production, il serait très intéressant de la remplacer par l'instanciation géométrique, qui permet de rendre plusieurs instances d'un même objet en un seul appel à la carte. Dans ce cas, le développement des règles devrait générer les matrices de transformation adéquates à

chaque élément terminal. Ensuite, l'évaluation des terminaux serait réalisée avec un appel à la carte par géométrie (au lieu d'un unique appel). Cette méthode implique donc un coût CPU plus élevé en contrepartie.

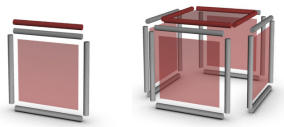
Comme indiqué dans la première contribution, le développement des règles est réalisé en une seule étape dans le GPU. Neanmoins, il serait très avantageux de pouvoir réaliser plusieurs passes de développement à la suite pour augmenter encore le parallélisme. En effet en prenant en entrée d'une passe N, les sorties de la passe N-1, on pourra aisément amplifier les graines d'entrée et permettre un meilleur parallélisme. Par exemple, un artiste pourrait dans une premiere étape générer des graines d'arbres ou de végétation sur terrain avec une grammaire de population, qui seront par la suite réellement générés en parallèle par d'autres grammaires. Un autre exemple consiste à ne traiter les branches filles que dans une seconde passe et ainsi éviter la divergence.

Notre pipeline utilise la fonctionnalité du *transform feedback* pour stocker les résultats intermédiaires au besoin sur GPU. Du fait de l'arrivée de cette fonctionnalité avec OpenGL|ES 3.0, notre approche pourrait être utilisée sur des matériels mobiles (téléphones, tablettes, etc). Ainsi il suffirait de transmettre les grammaires légères sur les réseaux à faible bande-passante et laisser les matériels clients se charger de la génération et de l'affichage des modèles complexes.

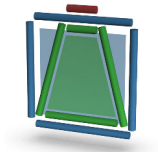
Pour finir nous avons introduit une méthode de *marching* qui effectue des expansions sur des maillages pré-générés. Notre approche pourrait etre étendue pour réaliser des opérations de *marching* sur des modèles procéduraux. Une solution serait de stocker une représentation simplifiée du modèle procédural et d'analyser ensuite cette représentation. Une seconde idée serait d'utiliser une approche de lancer de rayons directement de la grammaire, similairement à [MGHS11] pour calculer les surfaces générées à la demande.

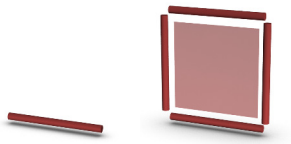
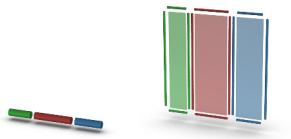
Appendix A

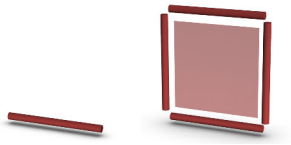
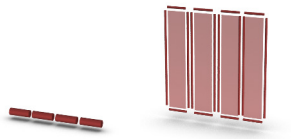
Grammar rule library

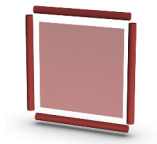
Pred $\rightsquigarrow$ Extrude(**v**, **height**, **scale**, **angle**){ *ExtrSucc*, *TopSucc* }	
▷ Parameters • *vec3* **v** : extrusion vector • *float* **height** : height of extrusion • *float* **scale** : scale of extrusion • *float* **angle** : twist angle ▷ Successors • *ExtrSucc* : extruded element successors • *TopSucc* : top element successor	Input primitives Output primitives 

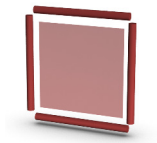
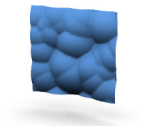
Pred $\rightsquigarrow$ Marching(**direction**, **stepSize**){ *quadSucc*, *segSucc* }	
▷ Parameters • *vec2* **direction** : 2D marching direction • *int* **stepSize** : marching size in pixels ▷ Successors • *quadSucc* : extruded element successor, containing information for geometry interpolation • *topSucc* : top segment successor	Input primitives Output primitives

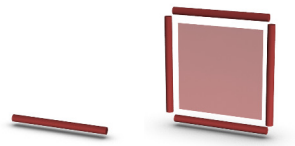
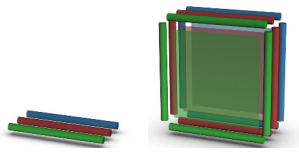
Pred $\rightsquigarrow$ Join(**height**, **v1**, **v2**){ *quadClippedSucc*, *quadSupportSucc*, *segTopSucc* }	
▷ Parameters • **height** : height of extrusion • **v1**; **v2** : target 2D segment ▷ Successors • *quadClippedSucc* : clipped element successor • *quadSupportSucc* : support element successor • *segTopSucc* : top segment successor	Input primitives Output primitives 

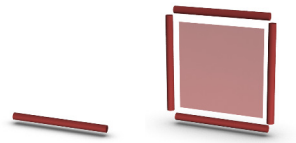
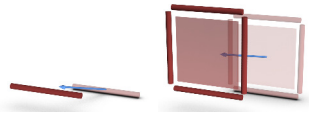
Pred $\rightsquigarrow$ Split(**axis**, **size**^{1}, **size**^{2}, ..., **size**$^{N-1}$){ *Succ*^{1}, *Succ*^{2}, ..., *Succ*^{N} }	
▷ Parameters • *char* **axis** : split axis X = horizontal Y = vertical • *float* **size**^{1} : first split size • *float* **size**^{2} : second split size • *float* **size**$^{N-1}$: last split size ▷ Successors • *Succ*^{1} : first element successor • *Succ*^{2} : second element successor • *Succ*^{N} : last element successor	Input primitives  Output primitives 

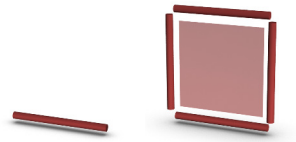
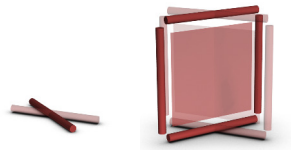
Pred $\rightsquigarrow$ Repeat(**axis**, **size**, **option**){ *Succ* }	
▷ Parameters • *char* **axis** : repeat axis X = horizontal Y = vertical • *float* **size** : repeat size • *int* **option** : repeat option 1 = last element fits the remaining space 2 = adapted size for equal repartition ▷ Successors • *Succ* $\rightarrow$ successor for each element	Input primitives  Output primitives 

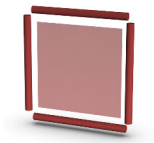
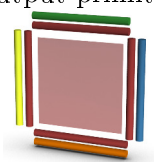
Pred $\rightsquigarrow$ Project(**axis^1**, **axis^2**){ *Succ* }	
▷ Parameters • *vec3* **axis^1** : first axis defining the projection plane • *vec3* **axis^2** : second axis defining the projection plane ▷ Successors • *Succ* : projected element successor	Input primitives  Output primitives

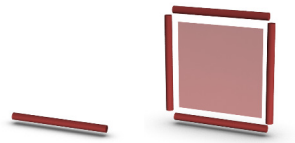
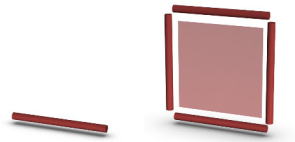
Pred $\rightsquigarrow$ Shape(**id**, **depthRatio**)	
▷ Parameters • *int* **id** : geometric texture id • *float* **depthRatio** : depth ratio of the geometry ▷ No Successors	Input primitives  Output primitives 

Pred $\rightsquigarrow$ Branch(**N**){ $Succ^1$, $Succ^2$, ... $Succ^N$ }	
▷ Parameters • **N** : number of branches ▷ Successors • $Succ^1$: first branch successor • $Succ^2$: second branch successor • $Succ^N$: last branch successor	Input primitives  Output primitives 

Pred $\rightsquigarrow$ Translate(**v**){ *Succ* }	
▷ Parameters • *vec3* **v** : translation vector ▷ Successors • *Succ* : element successor	Input primitives  Output primitives 

Pred $\rightsquigarrow$ Rotate(**origin**, **v**, **angle**){ *Succ* }	
▷ Parameters • *vec3* **origin** : center of rotation • *vec3* **v** : axis of rotation • *float* **angle** : angle of rotation ▷ Successors • *Succ* : element successor	Input primitives  Output primitives 

Pred $\rightsquigarrow$ Explode(){ $Succ_S^1$, $Succ_S^2$, $Succ_S^3$, $Succ_S^4$, $Succ_Q$ }	
▷ No Parameters ▷ Successors • $Succ_{Seg}^1$: left segment successor • $Succ_{Seg}^2$: front segment successor • $Succ_{Seg}^3$: right segment successor • $Succ_{Seg}^4$: back segment successor • $Succ_{Quad}$: initial element successor	Input primitives  Output primitives 

Pred $\rightsquigarrow$ Condition(**expression**){ $Succ^1$, $Succ^2$ }	
▷ Parameters • *bool* **expression** : expression to evaluate at run-time ▷ Successors • $Succ^1$: successor if expression is TRUE • $Succ^2$: successor if expression is FALSE	Input primitives  Output primitives 

Pred $\rightsquigarrow$ RotateDir(**angle**){ $Succ$ }	
▷ Parameters • *float* **angle** : angle of rotation for the marching direction ▷ Successors • $Succ$: element successor	Input primitives Output primitives

Bibliography

[AAAG96] Oswin Aichholzer, Franz Aurenhammer, David Alberts, and Bernd Gärtner. A novel type of skeleton for polygons. In *The Journal of Universal Computer Science*, pages 752–761. Springer Berlin Heidelberg, 1996.

[All13] Allegorithmic®. `http://www.allegorithmic.com`, 2013.

[Amd67] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[ARB07] Daniel G Aliaga, Paul A Rosen, and Daniel R Bekins. Style grammars for interactive visualization of architecture. *IEEE Transactions on Visualization and Computer Graphics*, 13(4):786–797, 2007.

[Aut13a] Autodesk®: 3dsmax®. `http://www.autodesk.fr/products/autodesk-3ds-max`, 2013.

[Aut13b] Autodesk®: Maya®. `http://www.autodesk.fr/products/autodesk-maya`, 2013.

[Aut13c] Autodesk®: Mudbox®. `http://www.autodesk.com/products/mudbox`, 2013.

[AYRW09] Saif Ali, Jieping Ye, Anshuman Razdan, and Peter Wonka. Compressed facade displacement maps. *IEEE Transactions on Visualization and Computer Graphics*, 15(2):262–273, 2009.

[Baj97] Chandrajit Bajaj. *Introduction to implicit surfaces*. Morgan Kaufmann, 1997.

[Ber66] Arthur J Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, 1966.

[BMG13] Cyprien Buron, Jean-Eudes Marvie, and Pascal Gautron. Gpu roof grammars. In *Eurographics 2013 Short Papers*, pages 85–88. The Eurographics Association, 2013.

[BMJ+11] Bedřich Beneš, Michel Abdul Massih, Philip Jarvis, Daniel G Aliaga, and Carlos A Vanegas. Urban ecosystem design. In *Proceedings of Symposium on Interactive 3D Graphics and Games 2011*, pages 167–174. ACM, 2011.

[BS05] Tamy Boubekeur and Christophe Schlick. Generic mesh refinement on gpu. In *Proceedings of Graphics Hardware 2005*, pages 99–104. ACM, 2005.

[BŠMM11] Bedrich Beneš, Ondrej Šťava, R Měch, and Gavin Miller. Guided procedural modeling. *Computer graphics forum, Proceedings of Eurographics 2011*, 30(2):325–334, 2011.

[BWS10] Martin Bokeloh, Michael Wand, and Hans-Peter Seidel. A connection between partial symmetry and inverse procedural modeling. *ACM Transactions on Graphics (TOG), Proceedings of ACM SIGGRAPH 2010*, 29(4):104, 2010.

[CC78] Edwin Catmull and James Clark. Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer-aided design*, 10(6):350–355, 1978.

[CEW+08] Guoning Chen, Gregory Esch, Peter Wonka, Pascal Müller, and Eugene Zhang. Interactive procedural street modeling. *ACM Transactions on Graphics (TOG), Proceedings of ACM SIGGRAPH 2008*, 27(3):103, 2008.

[CHCH06] Nathan A Carr, Jared Hoberock, Keenan Crane, and John C Hart. Fast gpu ray tracing of dynamic meshes using geometry images. In *Proceedings of Graphics Interface 2006*, pages 203–209. Canadian Information Processing Society, 2006.

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[Cho65] Noam Chomsky. *Aspects of the Theory of Syntax*, volume 11. The MIT press, 1965.

[DDSD03] Xavier Décoret, Frédo Durand, François X Sillion, and Julie Dorsey. Billboard clouds for extreme model simplification. *ACM Transactions on Graphics (TOG), Proceedings of ACM SIGGRAPH 2003*, 22(3):689–696, 2003.

[DF81] Frances Downing and Ulrich Flemming. *The bungalows of Buffalo*. Department of Architecture, Carnegie-Mellon University, 1981.

[DHL+98] Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomír Měch, Matt Pharr, and Przemyslaw Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. In *Proceedings of ACM SIGGRAPH 1998*, pages 275–286. ACM, 1998.

[Dua02] John Duarte. *Malagueira Grammar - towards a tool for customizing Alvaro Siza's mass houses at Malagueira*. PhD thesis, MIT School of Architecture and Planning, 2002.

[Edm60] Jack Edmonds. A combinatorial representation of polyhedral surfaces. *Notices of the American Mathematical Society*, 7, 1960.

[EMP+02] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2002.

[Esr12] Esri®: City engine®. `http://www.procedural.com`, 2012.

[FFC82] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, 1982.

[Fle87] Ulrich Flemming. More than the sum of parts: the grammar of queen anne houses. *Environment and Planning B: Planning and Design*, 14(3):323–350, 1987.

[Fly66] Michael J Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[FT99] Fabien Feschet and Laure Tougne. Optimal time computation of the tangent of a discrete curve: Application to the curvature. In *Discrete Geometry for Computer Imagery*, volume 1568, pages 31–40. Springer Berlin Heidelberg, 1999.

[GBP11] Gaël Guennebaud, Loïc Barthe, and Mathias Paulin. *Point-based graphics*, chapter Real-Time Refinement. Morgan Kaufmann, 2011.

[GG10] Li Gang and Shi Guangshun. Procedural modeling of urban road network. In *2010 International Forum on Information Technology and Applications (IFITA)*, volume 1, pages 75–79. IEEE, 2010.

[GGG+13] Jean-David Génevaux, Eric Galin, Eric Guérin, Adrien Peytavie, and Bedřich Beneš. Terrain generation using procedural models based on hydrology. *ACM Transactions on Graphics (TOG), Proceedings of ACM SIGGRAPH 2013*, 32(4):143:1–143:13, 2013.

[GGH02] Xianfeng Gu, Steven J Gortler, and Hugues Hoppe. Geometry images. *ACM Transactions on Graphics (TOG), Proceedings of ACM SIGGRAPH 2002*, 21(3):355–361, 2002.

[GPMG10] Eric Galin, Adrien Peytavie, Nicolas Maréchal, and Eric Guérin. Procedural generation of roads. *Computer Graphics Forum, Proceedings of Eurographics 2010*, 29(2):429–438, 2010.

[Gus88] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[HGA+10] Houssam Hnaidi, Eric Guérin, Samir Akkouche, Adrien Peytavie, and Eric Galin. Feature based terrain generation using diffusion equation. *Computer Graphics Forum, Proceedings of Pacific Graphics 2010*, 29(7):2179–2186, 2010.

[HWA+10] Simon Haegler, Peter Wonka, Stefan Mueller Arisona, Luc Van Gool, and Pascal Müller. Grammar-based encoding of facades. *Computer Graphics Forum, Proceedings of Symposium on Rendering 2010*, 29(4):1479–1487, 2010.

[IOI06] Takashi Ijiri, Shigeru Owada, and Takeo Igarashi. The sketch l-system: Global control of tree modeling using free-form strokes. In *Smart Graphics*, volume 4073 of *Lecture Notes in Computer Science*, pages 138–146. Springer Berlin Heidelberg, 2006.

[JWP05] Stefan Jeschke, Michael Wimmer, and Werner Purgathofer. Image-based representations for accelerated rendering of complex scenes. *STAR reports, Eurographics*, 2005:1–20, 2005.

[KE81] Hank Koning and Julie Eizenberg. The language of the prairie: Frank lloyd wright's prairie houses. *Environment and Planning B*, 8(3):295–323, 1981.

[KK11a] Lars Krecklau and Leif Kobbelt. Procedural modeling of interconnected structures. *Computer Graphics Forum, Proceedings of Eurographics 2011*, 30(2):335–344, 2011.

[KK11b] Lars Krecklau and Leif Kobbelt. Realtime compositing of procedural facade textures on the gpu. In *3DARCH11*, 2011.

[KPK10] Lars Krecklau, Darko Pavic, and Leif Kobbelt. Generalized use of non-terminal symbols for procedural modeling. *Computer Graphics Forum*, 29(8):2291–2303, 2010.

[KW11] Tom Kelly and Peter Wonka. Interactive architectural modeling with procedural extrusions. *ACM Transactions on Graphics (TOG)*, 30(2):14, 2011.

[KWm10] David B Kirk and W Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2010.

[LBZ+11] Yuanyuan Li, Fan Bao, Eugene Zhang, Yoshihiro Kobayashi, and Peter Wonka. Geometry synthesis on surfaces using field-guided shape grammars. *IEEE Transactions on Visualization and Computer Graphics*, 17(2):231–243, 2011.

[LCV03] Javier Lluch, Emilio Camahort, and Roberto Vivó. Procedural multiresolution for plant and tree rendering. In *Proceedings of the 2Nd International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, AFRIGRAPH '03, pages 31–38. ACM, 2003.

[LD03] Robert G Laycock and AM Day. Automatically generating large urban environments based on the footprint data of buildings. In *Proceedings of the eighth ACM symposium on Solid modeling and applications*, pages 346–351. ACM, 2003.

[LH04] Patrick Lacz and John C Hart. Procedural geometry synthesis on the gpu. In *Workshop on general purpose computing on graphics processors*, pages 23–31. ACM, 2004.

[LHP11] Luc Leblanc, Jocelyn Houle, and Pierre Poulin. Component-based modeling of complete buildings. In *Proceedings of Graphics Interface 2011*, pages 87–94. Canadian Human-Computer Communications Society, 2011.

[Lie94] Pascal Lienhardt. N-dimensional generalized combinatorial maps and cellular quasi-manifolds. *International Journal of Computational Geometry & Applications*, 4(03):275–324, 1994.

[Lin68] Aristid Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280 – 299, 1968.

[Loo87] Charles Loop. *Smooth subdivision surfaces based on triangles*. M.S. Mathematics thesis. Department of Mathematics, University of Utah, 1987.

[LWW08] Markus Lipp, Peter Wonka, and Michael Wimmer. Interactive visual editing of grammars for procedural architecture. *ACM Transactions on Graphics (TOG), Proceedings of ACM SIGGRAPH 2008*, 27(3):102, 2008.

[LWW09] Markus Lipp, Peter Wonka, and Michael Wimmer. Parallel generation of l-systems. In *Vision, Modeling, and Visualization Workshop (VMV) 2009*, pages 205–214, 2009.

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

[M+65] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[Mag09] Milán Magdics. Real-time generation of l-system scene models for rendering and interaction. In *Proceedings of the 25th Spring Conference on Computer Graphics*, pages 67–74. ACM, 2009.

[Män88] Martti Mäntylä. *An introduction to solid modeling.* W. H. Freeman & Co., 1988.

[MBG+12] Jean-Eudes Marvie, Cyprien Buron, Pascal Gautron, Patrice Hirtzlin, and Gaël Sourimant. Gpu shape grammars. *Computer Graphics Forum, Proceedings of Pacific Graphics 2012*, 31(7):2087–2095, 2012.

[McG11] Morgan McGuire. Efficient triangle and quadrilateral clipping within shaders. *Journal of Graphics, GPU, and Game Tools*, 15(4):216–224, 2011.

[MGHS11] Jean-Eudes Marvie, Pascal Gautron, Patrice Hirtzlin, and Gael Sourimant. Render-time procedural per-pixel geometry generation. In *Proceedings of Graphics Interface 2011*, pages 167–174. Canadian Human-Computer Communications Society, 2011.

[MP96] Radomír Měch and Przemyslaw Prusinkiewicz. Visual models of plants interacting with their environment. In *Proceedings of ACM SIGGRAPH 1996*, pages 397–410. ACM, 1996.

[MPB05] Jean-Eudes Marvie, Julien Perret, and Kadi Bouatouch. The fl-system: a functional l-system for procedural geometric modeling. *The Visual Computer*, 21(5):329–339, 2005.

[MSK10] Paul Merrell, Eric Schkufza, and Vladlen Koltun. Computer-generated residential building layouts. *ACM Transactions on Graphics (TOG), Proceedings of ACM SIGGRAPH ASIA 2010*, 29(6):181, 2010.

[MWH+06] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *ACM Transactions on Graphics (TOG), Proceedings of ACM SIGGRAPH 2006*, 25(3):614–623, 2006.

[MZWVG07] Pascal Müller, Gang Zeng, Peter Wonka, and Luc Van Gool. Image-based procedural modeling of facades. *ACM Transactions on Graphics (TOG), Proceedings of ACM SIGGRAPH 2007*, 26(3):85, 2007.

[Pet08] Jörg Peters. Pn-quads. Technical report, Technical Report 2008-421, Dept. CISE, University of Florida, 2008.

[Pix13] Pixologic®: Zbrush®. `http://pixologic.com/zbrush/`, 2013.

[PJM94] Przemyslaw Prusinkiewicz, Mark James, and Radomír Měch. Synthetic topiary. In *Proceedings of ACM SIGGRAPH 1994*, pages 351–358. ACM, 1994.

[PLH+90] Przemyslaw Prusinkiewicz, Aristid Lindenmayer, James S Hanan, F David Fracchia, Deborah R Fowler, Martin JM de Boer, and Lynn Mercer. *The algorithmic beauty of plants.* The virtual laboratory. Springer-Verlag, 1990.

[PM01] Yoav IH Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of ACM SIGGRAPH 2001*, pages 301–308. ACM, 2001.

[PMKL01] Przemyslaw Prusinkiewicz, Lars Mündermann, Radoslaw Karwowski, and Brendan Lane. The use of positional information in the modeling of plants. In *Proceedings of ACM SIGGRAPH 2001*, pages 289–300. ACM, 2001.

[PO08] Anjul Patney and John D Owens. Real-time reyes-style adaptive surface subdivision. *ACM Transactions on Graphics (TOG), Proceedings of ACM SIGGRAPH ASIA 2008*, 27(5):143, 2008.

[RV77] Aristides AG Requicha and Herbert B Voelcker. *Constructive solid geometry.* Technical memorandum. Production Automation Project, University of Rochester, 1977.

[ŠBM+10] Ondrej Št'ava, Bedrich Beneš, R Měch, Daniel G Aliaga, and Peter Krištof. Inverse procedural modeling by automatic generation of l-systems. *Computer Graphics Forum*, 29(2):665–674, 2010.

[SDKT+09] Ruben M Smelik, Klaas Jan De Kraker, Tim Tutenel, Rafael Bidarra, and Saskia A Groenewegen. A survey of procedural methods for terrain modelling. In *Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)*, 2009.

[SG71] George Stiny and James Gips. Shape grammars and the generative specification of painting and sculpture. In *IFIP Congress*, pages 1460–1465, 1971.

[SJP05] Le-Jeng Shiue, Ian Jones, and Jörg Peters. A realtime gpu subdivision kernel. *ACM Transactions on Graphics (TOG), Proceedings of ACM SIGGRAPH 2005*, 24(3):1010–1015, 2005.

[SM+78] George Stiny, William J Mitchell, et al. The palladian grammar. *Environment and Planning B*, 5(1):5–18, 1978.

[SP86] Thomas W Sederberg and Scott R Parry. Free-form deformation of solid geometric models. *Proceedings of ACM SIGGRAPH 1986*, 20(4):151–160, 1986.

[SPR06] Alla Sheffer, Emil Praun, and Kenneth Rose. Mesh parameterization methods and their applications. *Foundations and Trends® in Computer Graphics and Vision*, 2(2):105–171, 2006.

[SWG+03] Pedro V Sander, Zoë J Wood, Steven J Gortler, John Snyder, and Hugues Hoppe. Multi-chart geometry images. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, SGP '03, pages 146–155. Eurographics Association, 2003.

[Tho07] Thomas luft: Ivy generator. `http://graphics.uni-konstanz.de/~luft/ivy_generator`, 2007.

[TLL+11] Jerry O Talton, Yu Lou, Steve Lesser, Jared Duke, Radomír Měch, and Vladlen Koltun. Metropolis procedural modeling. *ACM Transactions on Graphics (TOG)*, 30(2):11, 2011.

[VAW+10] Carlos A Vanegas, Daniel G Aliaga, Peter Wonka, Pascal Müller, Paul Waddell, and Benjamin Watson. Modelling the appearance and behaviour of urban spaces. *Computer Graphics Forum*, 29(1):25–42, 2010.

[VKW+12] Carlos A Vanegas, Tom Kelly, Basil Weber, Jan Halatsch, Daniel G Aliaga, and Pascal Müller. Procedural generation of parcels in urban modeling. *Computer Graphics Forum, Proceedings of Eurographics 2012*, 31(2):681–690, 2012.

[VPBM01] Alex Vlachos, Jörg Peters, Chas Boyd, and Jason L Mitchell. Curved pn triangles. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 159–166. ACM, 2001.

[Whi06] Chris White. King kong: the building of 1933 new york city. In *ACM SIGGRAPH 2006 Sketches*, volume 6, page 96, 2006.

[WMWF07] Peter Wonka, Pascal Müller, Ben Watson, and Andy Fuller. Urban design and procedural modeling. In *ACM SIGGRAPH 2007 courses*, pages 229–229. ACM, 2007.

[WOD09] Emily Whiting, John Ochsendorf, and Frédo Durand. Procedural modeling of structurally-sound masonry buildings. *ACM Transactions on Graphics (TOG), Proceedings of SIGGRAPH ASIA 2009*, 28(5):112:1–112:9, 2009.

[WWSR03] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. *ACM Transactions on Graphics (TOG), Proceedings of ACM SIGGRAPH 2003*, 22(3):669–677, 2003.

[YHL+07] Tingjun Yang, Zhengge Huang, Xingsheng Lin, Jianjun Chen, and Jun Ni. A parallel algorithm for binary-tree-based string rewriting in l-systems. In *International Multi-Symposiums on Computer and Computational Sciences 2007*, pages 245–252. IEEE, 2007.

[ZHW+06] Kun Zhou, Xin Huang, Xi Wang, Yiying Tong, Mathieu Desbrun, Baining Guo, and Heung-Yeung Shum. Mesh quilting for geometric texture

synthesis. *ACM Transactions on Graphics (TOG), Proceedings of ACM SIGGRAPH 2006*, 25(3):690–697, 2006.

Abstract

With the increasing computing and storage capabilities of recent hardware, movie and video game industries desire huger realistic environments. However, modeling such sceneries by hand turns out to be highly time consuming and costly. On the other hand, procedural modeling provides methods to easily generate high diversity of elements such as vegetation and architecture. While grammar rules bring a high-level powerful modeling tool, using these rules is often a tedious task, necessitating frustrating trial and error process. Moreover, as no solution proposes real-time generation and rendering for massive environments, artists have to work on separate parts before integrating the whole and see the results.

In this research, we aim to provide interactive generation and rendering of very large sceneries, while offering artist-friendly methods for controlling grammars behavior. We first introduce a GPU-based pipeline providing parallel procedural generation at render time. To this end we propose a segment-based expansion method working on independent elements, thus allowing for parallel amplification. We then extend this pipeline to permit the construction of models relying on internal contexts, such as roofs. We also present external contexts to control grammars with surface and texture data. Finally, we integrate a LOD system with optimization techniques within our pipeline providing interactive generation, edition and visualization of massive environments. We demonstrate the efficiency of our pipeline with a scene comprising hundred thousand trees and buildings each, representing 2 terabytes of data.

Keywords: procedural modeling, interactivity, parallelism, graphics cards, massive virtual environments.

Résumé

Afin de créer des productions toujours plus réalistes, les industries du jeu vidéo et du cinéma cherchent à générer des environnements de plus en plus larges et complexes. Cependant, la modélisation manuelle des objets 3D dans de tels décors se révèle très coûteuse. A l'inverse, les méthodes de génération procédurale permettent de créer facilement une grande variété d'objets, tels que les plantes et les bâtiments. La modélisation par règles de grammaire offre un outil de haut niveau pour décrire ces objets, mais utiliser correctement ces règles s'avère très souvent compliqué. De plus, aucune solution de modélisation basée grammaire ne supporte l'édition et la visualisation d'environnements massifs en temps interactif. Dans un tel scénario, les artistes doivent modifier les objets en dehors de la scène avant de voir le résultat intégré.

Dans ces travaux de recherche, nous nous intéressons à la génération procédurale et au rendu d'environnements à grande échelle. Nous voulons aussi faciliter la tâche des artistes avec des outils intuitifs de contrôle de grammaires. Tout d'abord nous proposons un système permettant la génération procédurale en parallèle sur le GPU en temps interactif. Pour cela, nous adoptons une approche d'expansion indépendante par segment, permettant une amplification des données en parallèle. Nous étendons ce système pour générer des modèles basés sur une structure interne, tels que les toits. Nous présentons aussi une solution utilisant des contextes externes pour contrôler facilement les grammaires par le biais de surface ou de texture. Pour finir nous intégrons un système de niveaux de détails et des techniques d'optimisation permettant la génération, l'édition et la visualisation interactives d'environnements à grande échelle. Grâce à notre système il est possible de générer et d'afficher interactivement des scènes comprenant des milliers de bâtiments et d'arbres, représentant environ 2 teraoctets de données.

Mots-clés: modélisation procédurale, interactivité, parallélisme, cartes graphiques, environnements virtuels massifs.