David Mandl

Automatic Generation of Surrogate Terminals for Shape Grammar Derivation

MASTER'S THESIS

to achieve the university degree of
Diplom-Ingenieur

Master's degree programme
Computer Science

submitted to

Graz University of Technology

Thesis supervisors

Dr. Markus Steinberger
Dr. Denis Kalkofen
Prof. Dr. Dieter Schmalstieg

Institute for Computer Graphics and Vision

Graz, Austria, Mar. 2016

Abstract

Today's computer graphic applications need a lot of high quality models that are usually generated by the work of many artists. Shape grammars are an approach to automatically generate these models based on a set of rules. PGA (parallel generation of architecture) is a system to evaluate shape grammars for models of buildings on the GPU. The PGA system evaluates the grammar and generates the geometry in parallel on the GPU.

In this work we show how to derive so called surrogate terminals from a given shape grammar. These terminals are used to replace otherwise generated geometry by textures that approximate the details of the skipped geometry. The textures replace the details on different detail levels. To derive the LOD levels an error metric between geometry and detailed surrogate texture is used. This approach also deals with randomness, thus grammars that are parametrized with random numbers.

Keywords. shape grammar, parallel, architecture, GPU

Kurzfassung

In heutigen Computergrafik-Anwendungen werden sehr viel 3D Modelle von hoher Qualität benötigt welche aufwendig von vielen Designern erstellt werden müssen. Shape Grammars sind eine Möglichkeit diese Modelle anhand von einer Menge Regeln automatisch zu erzeugen. PGA (Parallel Generation of Architecture) ist ein System das Shape Grammars verwendet um Gebäude zu erzeugen. PGA führt die Grammatik auf der Grafikkarte (GPU) aus um die Vorteile der Massiven Parallelen Datenverarbeitung auf der GPU zu nutzen (CUDA).

In dieser Arbeit geht es um die Ableitung sogenannter Surrogate Terminals für Shape Grammars. Diese Terminals werden verwendet um durch die Grammatik erzeugte Geometrie durch Texturen zu erstzen welche die ersetze Geometriy approximieren. Diese Texturen werden für verschiedene Detailstufen erzeugt. Um eine Bedingung für die verwendung der Detailstufen abzuleiten wird ein Fehlerkriterium verwendet um den Fehler zwischen 3D Geometrie und der Surrogate Texture zu bestimmen. Es können auch Zufallszahlen zur parametrisierung der Grammatik verwendet werden.

Statutory Declaration

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

The text document uploaded to TUGRAZonline is identical to the presented master's thesis dissertation.

Place	Date	Signature

Eidesstattliche Erklärung

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

Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Ort	Datum	Unterschrift

Acknowledgments

I would like to thank the following people.

My supervisor Markus Steinberger for all his feedback and suggestions. He has been very patient and always had an answer to my questions. Also thanks to him i can now work for the ICG. Mark Doktor, Pedro Poechat and Michael Kenzel, who laid out the foundation of the PGA framework which is the base of my work. As well as for their helpful suggestions and discussions. I also want to thank Denis Kalkofen for his helpful feedback and motivation to carry on with my work.

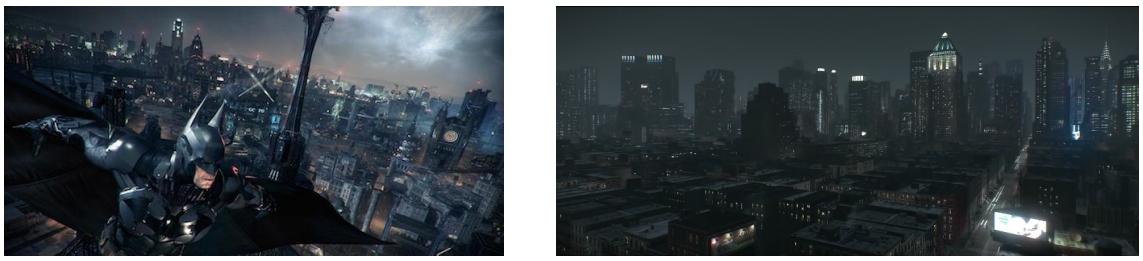
Contents

List of Figures

List of Tables

1 Introduction

Computer graphic application nowadays require a great amount of content to present high detailed virtual scenes to their users. These 3D environments consist of many 3D objects like buildings, characters and small objects. Figure 1.1 shows example renderings of state of the art games. In these examples large cities with hundreds of building are rendered by the game's graphics engine. With increasing hardware capabilities, rendering many detailed 3D objects is not much of a problem nowadays.



(a) **(b)**

Figure 1.1: Example screenshots from state of the art games. (a) Batman flying over Gotham City. From Batman Arkham Knight ©Rocksteady (b) Tom Clancy The Division takes place in a large city ©Ubisoft.

To make such big 3D scenes a large number of 3D objects is needed. All these objects are often still created by hand by 3D artists. These artists still rely on the same 3D modeling programs from years ago. Tools like Maya [1] and Blender [2] are still used to produce the necessary 3D assets for characters, props and buildings. These tools are not designed for mass production of 3D models, i.e. authoring hundreds of 3D objects.

[1] `http://www.autodesk.de/products/maya`
[2] `http://www.blender.org`

However, many of these assets could be generated automatically so the artists could focus on other things in his workflow. For example modern video games like Far Cry 4 or Grand Theft Auto 5 offer a huge amount of content for its gamers. The creation of all the assets needed often requires many artists to spend hundreds of hours working on building the 3D models only. Many of these objects often show variations of basic objects that could be created automatically. For example city buildings often share a basic type of shape.

Another example is the movie industry, where nowadays a lot of scenes are done in green screen and Computer Generated Imagery (CGI) is used to represent certain scenes (background). These images can be either painted or created using 3D renderings. These scenes are usually created by 3D artists in many hours of work but most of these rendered scenes are only a few seconds on screen. These could be generated automatically to let the artists concentrate on detailed *CGI* that is used for creatures or other detailed models.

Many other fields exist which can benefit from the automatic creation of computer graphic content. For example architecture, computer aided design, medicine, etc. All share the same problem, a large amount of content has to be created which often include variations of basic shapes.

Shape grammars allow to greatly reduce the time and effort required to create 3D content. They provide a way to create assets like buildings procedurally, i.e. without any manual interaction. These grammars consist of a set of rules which determine how the geometry is automatically created. This way, only the rules need to be defined to produce the geometry for, e.g., buildings. Therefore, no interaction is needed to create the actual geometry. Another benefit is that any changes in the shape grammar can be viewed instantly by rendering the resulting geometry. With randomization whole cities can be created in a few seconds while it would take several work hours to create all those buildings by hand. See figure 1.2 for an example of a city generated by rules using the *PGA* system.

1.1 Problems

In shape grammars a problem is how the different content is generated. Either multiple grammars are needed or an attributed grammar in which the parameters allow the user to alter the results the grammar produces. However, the creation of a shape grammar is not as intuitive as creating geometry by hand. Therefore an editor is usually needed to see the resulting geometry of a shape grammar.

When generating multiple objects with a shape grammar an obvious problem

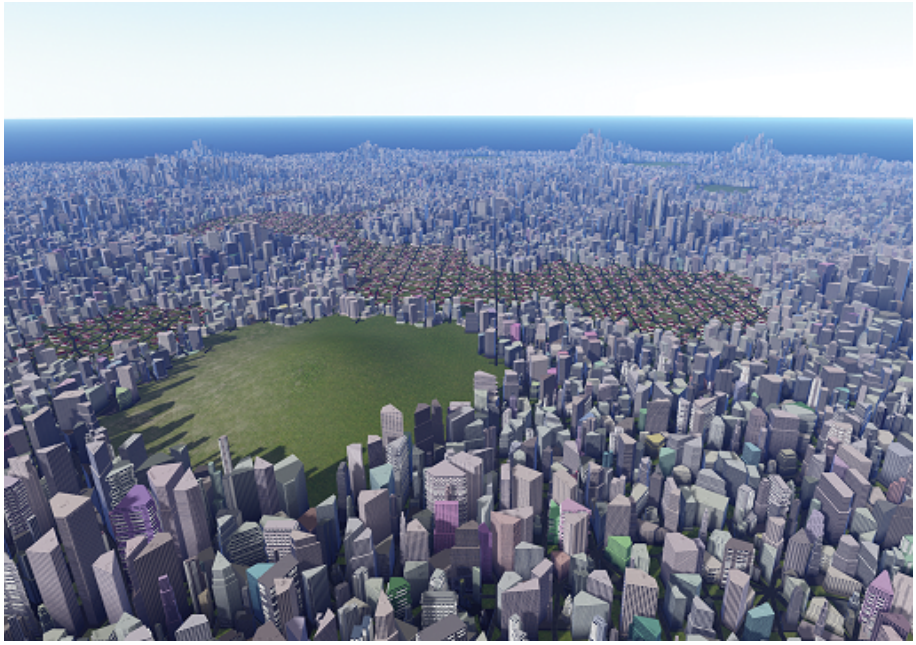


Figure 1.2: city buildings created with PGA

is that the different outcome shapes are always the same. To overcome this there are attributed grammars which can be parametrized to allow for different resulting shapes. To create a diverse city of many different buildings it would still be necessary to define every building with a set of grammar rules. A method to reduce the number of rules needed are random parameters that can be used to create many different buildings from a single shape grammar rule set.

Evaluation of the grammar can be done either on the Central Processing Unit (CPU) or on the Graphics Processing Unit (GPU). Evaluating grammars for many buildings on the *CPU* can be very slow because the grammar has to be processed sequential. The *GPU* is a lot faster in processing small tasks in parallel. The problem is that the grammar rules are dependent on their parent rules so the rules cant be easily executed in parallel. To overcome this issue, an evaluation system on the GPU has to introduce parallelism into the dependency graph that represents the parent-child relationship between generated shapes.

Another problem is that the shape grammar gets executed for every building even if its not visible or to far away to see any details. To overcome this one can consider the distances of the buildings to the camera and adjust the evaluation of the grammar to only generate enough detail for rendering. The missing detail could be replaced by an imposter rendering that resembles the geometry. For example a texture could be generated beforehand that resembles the replaced geometry on at a specific distance to

the camera.

1.2 Purpose

Using only a small set of rules shape grammars can produce a large amount of geometry automatically. While shape grammars can be evaluated either on the *CPU* or on the *GPU* for this work we focus on the evaluation on the *GPU* which has the advantage of having the geometry information directly available on the *GPU* for rendering.

Shape grammars work on geometric shapes, in this work we use simple shapes like boxes and quads. These shapes can be used to derive shapes of simple buildings. The shapes are defined in the shape grammar definition or rule-set. During the grammar evaluation these shapes are used to derive the geometry of every rule. The grammar evaluation stops at so called terminal symbols [23] where the actual geometry is produced. This is the point where the shape's geometric transformation comes into play to create the final output shape. Which shapes are used depends on the 3D models that needs to be created.

In this work, we explore the notion of so-called surrogate terminals [18] for shape grammars, which are used to replace detailed geometry by much simpler geometry during the evaluation of the shape grammar. This allows to stop the evaluation earlier and replace covered geometry. This means that they need to resemble the geometry of the replaced detail. To achieve this goal we present a derivation method to preprocess a shape grammar and create surrogate terminals for a given shape grammar.

To replace the detail, a custom texture has to be created that contains the replaced details. These textures are called surrogate textures. To render this textures it is also necessary to replace the geometry with a coarse shape that can be texture mapped with the surrogate texture.

Every surrogate terminal can be viewed as a *LOD*. To use multiple level of detail a condition is needed to derive the correct detail level at any time. Since the amount of detail that is rendered depends on the distance from the geometry to the camera, this distance can be used to derive the current needed detail level and choose the corresponding surrogate texture. In this way the optimal detail level can be chosen during runtime. To get the right camera distance for every detail level a metric is needed for comparing the detailed rendering with the surrogate terminal rendering. This metric need to minimize the error between these two renderings to get the right camera distance for switching the detail level.

Shape grammars can be parametrized to achieve different resulting shapes

using the same rule. These parameters can also be chosen randomly. In this work we describe how to incorporate random parameters in *GPU* shape grammars and how to use them during runtime. For surrogate terminals, this means that the replaced detail can be very different if random parameters are used. This can result in an large amount of different shapes and textures on terminal symbols. Thus, we also show how to cope with random parameters when creating surrogate textures and how to reduce the number of textures needed. And thus also reduce the amount of memory needed for storing the textures.

In this work, we tackle two major challenges associated with surrogate terminal creation: First, we show how to derive a surrogate texture that replaces the detailed geometry as closely as possible. Second, we tackle the problem of deriving the right distances for switching detail levels and how to compare the resulting rendering to the original detail rendering so that the error between both is as minimal as possible.

In this work the *PGA* system is used to evaluate shape grammars. *PGA* evaluates the grammar on the *GPU* using a rule scheduling mechanism. *PGA* uses the grammar definition to generate simple geometry based on shapes like boxes and quads. The system is implemented in C++ while using Compute Unified Device Architecture (CUDA) for the grammar evaluation part and OpenGL for the rendering part. The grammar definition is done with template meta programming using C++ templates. This system is extended with new features to allow for rendering of textured models and new parameters that allow random values. These features are needed to support the surrogate terminals.

1.3 Organization of this work

In chapter 2 we give an overview of related work in the field of shape grammars, procedural modelling and *GPU* scheduling. Also a short overview of level of detail methods is given. Chapter 3 introduces the existing *PGA* system and the needed extensions to support surrogate terminals. Also the concept behind surrogate terminals and surrogate textures are explained as well as their creation process. This chapter also deals with the automatic derivation of the *LOD* levels and presents a method to measure the quality of the created textures. In chapter 4 the implementation of needed extension to the *PGA* system is discussed and the creation process of surrogate terminals and their textures are explained. Along those lines we discuss how the image quality is measured and the detail levels are derived. Then, in chapter 5 we compare different scenes of buildings generated by *PGA* with and without surrogate terminals. The full quality between original rendering and surrogate rendering is also discussed in this chapter. Finally, in chapter 6 we conclude our work and give hints on possible future expansions for this system.

2 Related Work

This chapter reviews the work in the field procedural modeling, shape grammars and the approaches to evaluate these grammars and different methods for *LOD* rendering. The implementation of this work is based on the *PGA* framework [19] this work is also reviewed.

2.1 Procedural Modelling

Procedural modeling is a method to automatically create geometry for computer graphics. Methods for generating this geometry ranges from shape grammars [12] to programming languages [2]. The user defines a production process that is interpreted by a system that generates the geometry based on the input given. This allows to create parametrizable 3D models within short time compared to modeling those models in a program like blender [1].

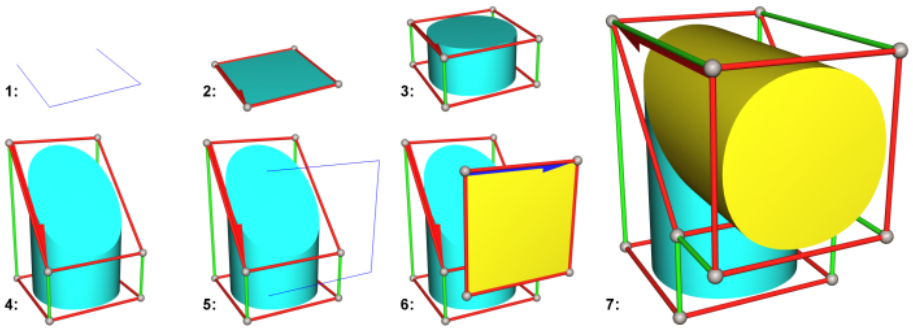


Figure 2.1: Using GML a user/designer defines a sequence of modeling operations to generate more complex geometric objects. In this example, a polygon is created to instantiate a cylinder, which is then cut and connected to another cylindric form.. Image from [2]

[1] `http://www.blender.org`

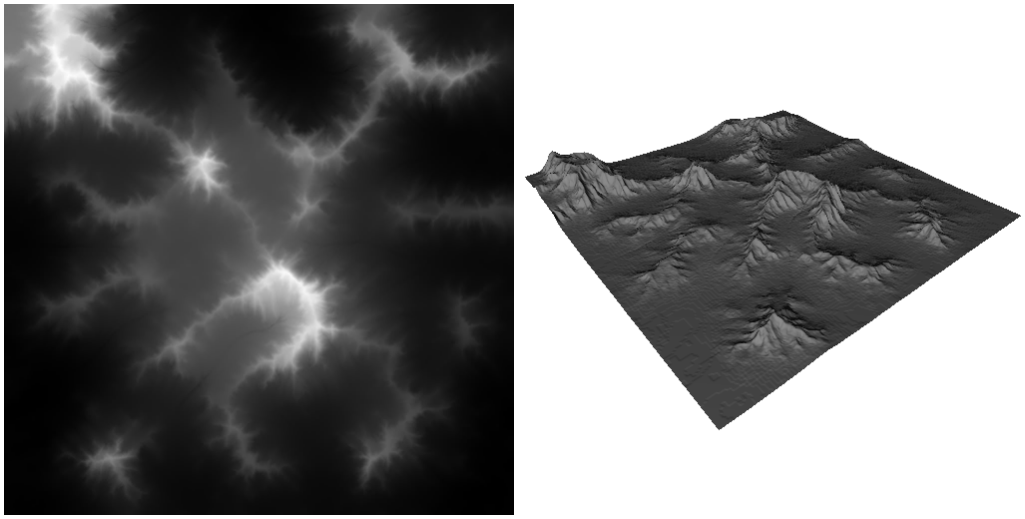
2.1.1 Procedural Generation of Textures

It is also possible to generate textures procedurally. One method to create such textures are noise patterns which are used to generate seemingly random textures. A popular approach to create such textures is to use so called Perlin noise [14]. These textures are used to perturb textures to create effects that seem to be randomly generated.

The application of procedural textures are for example particle systems (fire, rain, etc.), terrain modeling and simulating the sky and clouds.

2.1.2 Fractals

Fractals are a mathematical set with self similar patterns repeated on different scales. Many natural phenomena show these patterns, for example, plants, mountain ranges, river networks, etc. This is why fractals can be used to replicate these natural phenomena in computer graphics. In computer graphics, fractals are, e.g., used to create the height maps in procedural terrain approaches. [13].



(a) **(b)**

Figure 2.2: Heightmap (a) image generated by a fractal generator. Bright areas show high section (mountains) and dark areas are valleys. Terrain (b) Rendering of the generated terrain based on the heightmap.

2.1.3 L-Systems

L-Systems are a formal system that is used to describe the growth of plants. A formal grammar is used to define the production rules that form a string from an alphabet. This

system can be used for generating 3D plant models [15]. Due to the parallel rewriting of rules in L-Systems the derivation process can be executed in parallel on a *GPU* [7]. An example of trees generated by such a system can be seen in Figure 2.3. These trees were generated by the PGA system.

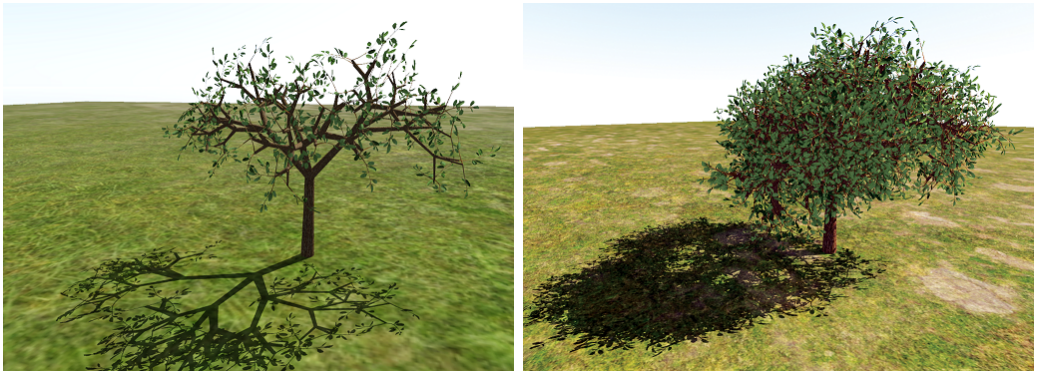


Figure 2.3: Trees generated by a L-System (PGA). Images from [19])

2.2 Shape Grammars

Shape grammars are a set of rewriting rules for geometric shapes. They were first introduced by G. Stiny [20]. Later he introduced the notion of so-called set grammars [22]. Set grammars are shape grammars with the difference that the rewriting of rules is done on an active set of symbols. In the beginning this set is populated with all starting symbols of the shape grammar. Then each symbol is evaluated and the resulting terminal or non-terminal symbol is placed in the set. This continues until the set contains only terminal symbols.

Shapes are the geometric entities that are produced and transformed by shape grammars. Every rule defined in the grammar has a shape that defines the geometry input for this rule. The rule itself defines the output or possible multiple output shapes that are generated after executing the rule. The basic definition of shapes is given by [21]:

"A shape is a limited arrangement of straight lines in three dimensional Euclidean space."

Since this is a very broad definition there is the notion of basic shapes which are a subset of shapes describing simple shapes. Working with lines would not be very intuitive, thus basic shapes are used since they can by easily translated to point sets,

lines or faces when needed. Examples for basic shapes are cubes, cylinders, spheres or prisms. These basic shapes can be parametrized and labelled.
A shape grammar consists of four sets [23]:

- a set of terminal symbols $N \subseteq U$
- a set of non-terminal symbols $T \subseteq U$
- a set of starting symbols $I \subseteq N$
- a set of production rules $R \subset U \times U^*$

Terminal symbols are symbols where the grammar evaluation stops, which is the point where the actual geometry is created. Non-terminal symbols are symbols which can be evaluated by using the appropriate production rule to derive the next set of symbols. The starting symbols are non-terminal symbols where the grammar evaluation starts. The production rules define how the left side which is always a non-terminal is replaced by a set of symbols. The grammar evaluation stops if there are only terminals symbols left.

Based on set grammars Wonka et al. [23] et al. introduced split grammars which work on basic shapes using so called split operations. Split operations allow to subdivide the current shape in smaller shapes that are contained in the current shape. Split grammars were applied to model architecture of buildings by Mueller et al. [12]. In this work city buildings are created by shape grammars which are applied on starting lots. See figure 2.4 for an example of buildings created with a split grammar.

.

Figure 2.4: Buildings generated with a split grammar. Image from [23]

Shape grammar rules define a geometric transformation between input and output shape.

These operations are based on L-systems [15].

To define the rules necessary to create a geometric shape, several systems have been proposed. For example, The Generative Modelling Language (GML) defines a stack based language that is interpreted to create geometric shapes [2]. Most systems that are based on shape grammars use a set of strings to define the rewriting rules.

The G^2 language derives its grammar rules from the Python programming language [4]. The city engine uses *CGA* shape which is a shape grammar for façade modelling [12]. See Figure 2.5 for example buildings created with *CGA*. Most prominent one is *CGA* Shape which executes its shape grammar on the *GPU*. See figure 2.5 and for example renderings of *CGA*.

.

Figure 2.5: Images show generated building with *CGA*

Besides the generation of buildings, shape grammars have been successfully used to produce high quality models. For example Synthetic texture can be produces to approximate the geometry of façades of buildings [5]. These textures are rearranged from source images of possible façades. See Figure 2.6 for an example of generated houses with synthetic put together façade textures.

.

Figure 2.6: Buildings with façades generated with by-example synthesis. Image from [5]

Model synthesis is an approach to generate complex models based on user defined input models [11].

(a) **(b)**

.

Figure 2.7: User created model (a) shows a user created model of an oil platform. Generated model (b) shows a complex model of oil platforms generated from (a) by model synthesis. Images from [11]

With structure preserving re-targeting [6] complex building models can be generated, which resemble the structural style of a given input architectural model. See figure 2.8 for example models generated with this method.

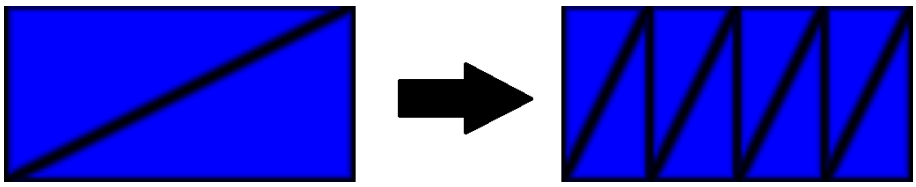
.

Figure 2.8: Building models using the photo at left bottom corner as input for the structural style. Image from [6]

2.3 Evaluating Shape Grammars

To create the actual geometry, shape grammars need to be evaluated. Which means that every rule is executed and the left symbol is replaced by either a terminal symbol or a non-terminal symbol. Terminal symbols are symbols where the grammar evaluations stops while non-terminal symbols are evaluated and rewritten depending on the rule that applies to this symbol [20].

The production rules in shape grammars are composed by a shape and an operator. The shape is the geometric shape on which the operator is executed. The operator defines a transformation that is applied on this shape. The resulting output shape depends on the operation applied. See 2.9 for a example of a repeat operation on a simple quad. Naturally this means a grammar is evaluated sequentially. However when a



.

Figure 2.9: Example for a repeat operation on a quad shape. The quad shape is subdivided in four equal sized quads along the x axis.

non-terminal symbol produces multiple non terminal symbols from a rewriting rule the grammar can be parallelized. Therefore, shape grammars have been evaluated on the CPU [12] and *GPU* [7].

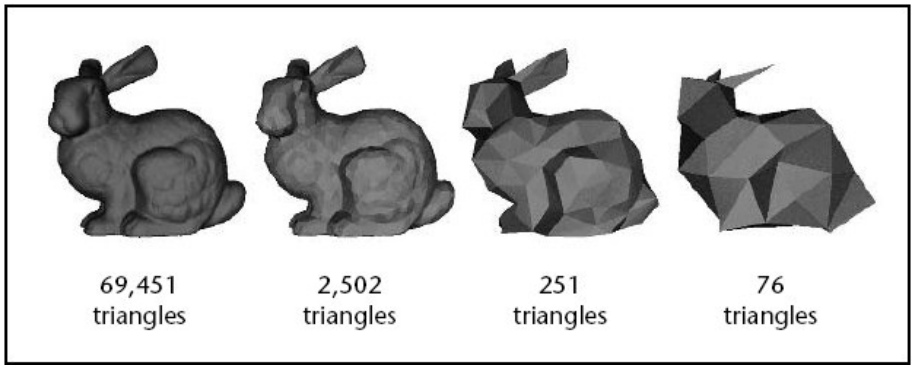
*GPU*s offer massive parallel processing power and have the advantage of rendering the result directly without the need of transferring it from the main memory

to the *GPU*. Lipp et al. [7] were one of the first to use the general purpose cores of a modern GPU to evaluate the grammar. They used *CUDA* to launch multiple kernels, each for every grammar symbol. A thread is launched for every non-terminal symbol in the active set. The kernel executes the operation on the current shape and produces the resulting output symbol which is added to the active set. This means, for each symbol a thread is launched to execute the operation of the current rule.

Several other work has been done regarding generation of buildings using shape grammars on the CPU and *GPU*. For example, Yang et al. [24] evaluate grammars based on L-systems on CPU clusters. Another approach to evaluate grammars on the *GPU* was presented by Magdics et al. [9]. They show how to use multiple render passes to generate intermediate symbols. The problem with such an approach is that many rendering passes are required the evaluate the shape grammar.

2.4 Level of Detail

Level of detail is an approach to use simplified 3D models to reduce the amount of rendered geometry and therefore increase the frame rate. Lower models replace the original detailed geometry when a certain condition is fulfilled, mostly the current distance to the users viewpoint. These simplified models can be created by hand using a 3D modeling program or automatically by mesh simplification for example using vertex decimation [16]. The two most common approaches for implementing *LOD* are discrete *LOD* and continuous *LOD* [8].



.

Figure 2.10: Example of different *LOD*s of the stanfort bunny

Discrete *LOD* simply replaces the detailed models with simplified ones when a certain distance threshold is reached. This has the advantage that the detail levels can be created offline. During rendering the runtime only has to pick the right *LOD*. The drawback is that it cannot be used with drastic simplification because the occurring pop

up would be easily noticeable.

Continuous *LOD* on the other hand derives the level of detail during runtime. This way, the needed detail level can be created exactly and not approximated by a few discrete models. This reduces the amount of pop up drastically and results in a much better granularity.

In this work the detail levels are created as a preprocessing step. During runtime the correct level is chosen based on a shape grammar operator that chooses the right level at a given distance. Thus using a discrete *LOD* approach.

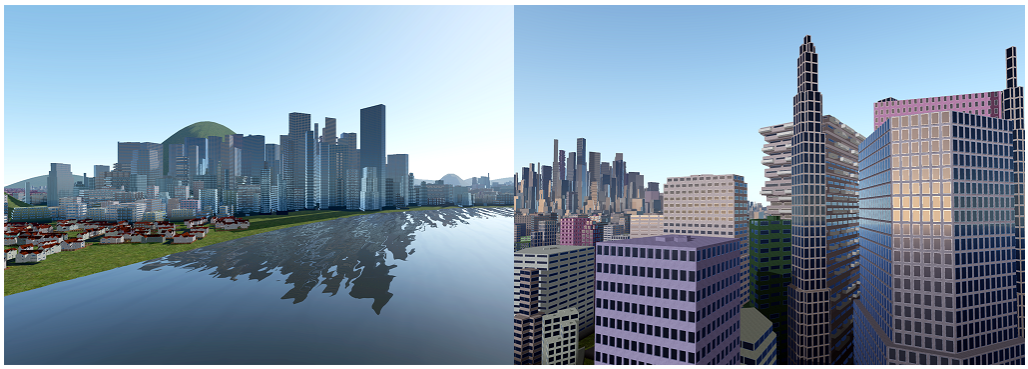
Level of detail approaches have also been used in procedural modelling using shape grammars. In the work of [1] the shape grammar is use to automatically generate simplified shapes of terminal symbols to be used for *LOD*. Since the detail levels are calculated for terminal shapes, the grammar evaluation will not change. While on contrast in this work we create *LOD* textures that can replace whole sub-trees in the derivation tree thus stopping the grammar evaluation earlier. The surrogate textures are generated using render-to-texture methods to create an image-based impostor [3].

To compare original and surrogate renderings the HDR-VDP-2 metric [10] is used. The HDR-VDP-2 compares the visibility as well as the image quality between two images. The result is a quality measure describing the similarity between the two images.

2.5 Parallel Generation of Architecture

Foundation of this work is the work on *PGA* [19] and on-the-fly rendering with the same system [18]. They are based on the Softshell framework for *GPU* scheduling [17]. The new *PGA* System that is used as an implementation foundation for this work, is using its own queuing framework to execute the workload on the *GPU*.

PGA is a system to automatically generate geometry for many buildings based on a shape grammar. The definition of these buildings is done by a shape grammar. A building is represented by a set of grammar rules. *PGA* evaluates these rules on the *GPU* in a parallel manner. This allows to generate and render those buildings during runtime [18]. The old version of *PGA* uses the Softshell framework to schedule and evaluate the grammar rules. Since then *PGA* has been reworked with a new queuing framework for *GPU* scheduling and pre-compiled rules to speed-up the evaluation of the grammar. The new *PGA* system is the foundation of this work and is extended to support the creation of surrogate terminals and textures. In Figure 2.11 you can see examples of generated buildings with the *PGA* system.



(a) **(b)**

Figure 2.11: Images show building generated with *PGA*. Image from [18]

3 Concept

PGA is an approach to evaluate shape grammars on the *GPU* for the creation of city buildings. Instead of modeling all the geometry manually in a modeling program, *PGA* generates it based on a given set of grammar rules. These rules are defined by the current shape (scope) and an operator that defines a geometric transformation. The shape is a simple geometric shape like a box or a quad. The transformation defines how the current shape is changed to create the geometry for the following rules.

3.1 Foundation

3.1.1 Shape Grammar

The shape grammar for *PGA* is based on *CGA* shape and uses split operations from split grammars [23]. Each rule defines a operation that is applied on a given input shape and a successor rule that is called after the current rule is finished. The output shapes of the current operation are the input shape(s) of the following rule. Additionally rules can be parameterized for example the size of the repetition extend when using a repeat operation.

There are different operator types in *PGA*:

- basic affine transformations: Translate, Rotate and Scale
- terminal operators: Generate, Discard
- changing the shape dimension: Extrude, ComponentSplit
- subdividing the shape (split operations): Repeat, Subdivide
- conditional production: IfSizeLess

The affine operations describe a simple geometric transformation which is applied to the current shape. The terminal operators define the creation of the actual geometry,

depending on the current shape a set of vertices, face and normals is created to represent this shape. The discard operator is needed when the final shape is skipped, this can be useful for creating windows in buildings. To change the shape type there are the Extrude and ComponentSplit operations. Extrude takes a 2D shape and extrudes it along the face normal to create a 3D shape. For example, a quad can be extruded to create a box shape. The ComponentSplit splits the shape into smaller parts, for example a box can be split into 6 quads for each box side. The IfSizeLess operator allows conditional production, for example when a subdivision operation is applied below a specified size and above this size a repeat operation is to fill the remaining size of the parent shape.

The split operators produce multiple output shapes, which is different to the other operators. Meaning a split rule will create multiple instances of the following rule that is evaluated. The operators work in a given scope, which is defined by the current shape of the object. Shapes are geometric primitives like quads, cubes or polygons. Some operators can change the current shape into another shape type. For example the "ComponentSplit" operator can split a box shape into six quad shapes. Most other operators work with the same shape as parent and child scope. The split operators can produce multiple output shapes from a single input shape.

These operators can be parametrized depending on the type of operation. For example the Repeat operator has the repetition extend as parameter. This value defines the size of every repetition extend which implies the number of repetitions. The basic *PGA* system supports constant values as parameters.

3.1.2 Derivation Tree

The grammar derivation tree represents the evaluation of the shape grammar in graph form. The starting rule is the root of the tree, the non-terminal symbols are the intermediate nodes and the terminal symbols represent the leaf nodes. The rules are represented by the edges between nodes. See figure 3.1 for a simple example of a derivation tree.

The blue nodes are non-terminal symbols and the green ones terminal symbols. The numbers represent the number of output symbols created by a rule. During the grammar evaluation this tree is traversed starting from the root node. The derivation tree is an important concept that is used in the next chapter to generate surrogate textures.

3.1.3 Generation

The evaluation of the grammar is done on the *GPU*. The grammar rules are the input and the output is a set of buffers to render the created scene. The evaluation of the rules is done using a queueing framework which balances the workload on the *GPU*. The grammar is evaluated starting with the first rule until the terminal symbols are reached.

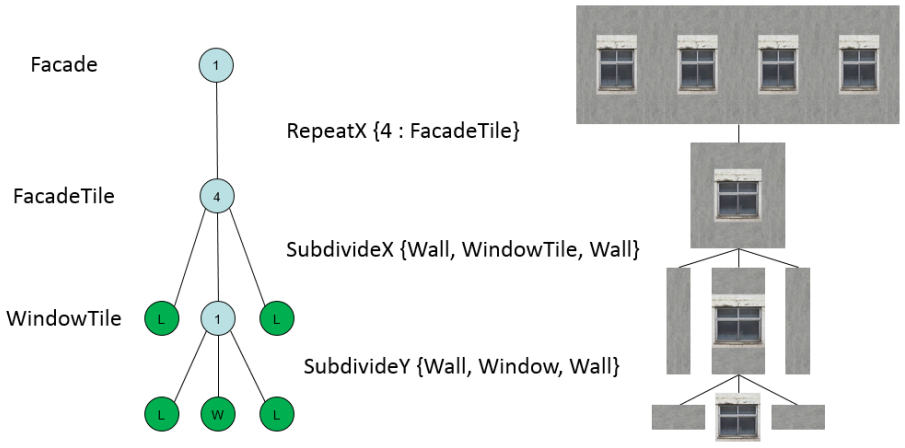


Figure 3.1: This example shows how a quad shaped façade is first split up in 4 tiles by the repeat operator. Then the subdivision operation is used to subdivide this tile along X and Y direction in wall and window tiles. Finally the wall and window tiles use the generate operator to produce the actual geometry.

The terminal rules generate the geometry given by the current shape and place it in so called terminal buffers. These buffers are used to render the actual geometry on the *GPU*.

To optimize the grammar evaluation the definition of grammar rules is done using template meta programming. Thus the generation of production rules is done during compile time which speeds up the whole process when the grammar is evaluated. Since the *GPU* is very efficient for processing small threads in a massively parallel manner it is important to schedule the rules accordingly to keep the *GPU* occupied. There are two different scheduling methods for the generation process:

Rule-based scheduling means that the grammar rule-set is pre-compiled. The rule-set is given as a set of nested templates so the compiler actually builds the grammar derivation tree during compile time. During runtime the tree is traversed and the operators are executed on the *GPU*.

Operator-based scheduling is a method to interpret the rule-set during runtime. To do this a dispatch table is build during runtime which contains all the rule information. Each rule has one entry in this lookup table. The advantage of this method is that rules can be changed during runtime without the need of compiling everything again.

3.1.4 Rendering

Rendering is done with OpenGL Application Programming Interface (API). The terminal buffers that are filled by the queueing framework are used to renderer the geometry

with appropriate shaders. Since there are multiple buffers for vertices, indices and texture coordinates the geometry can be rendered using simple texture mapping which is the basis for creating levels of detail later. There are different methods to render the resulting geometry in *PGA*.

Non-Instanced rendering means that the whole geometry (vertices, indices, texture coordinates) that is held in buffers. The disadvantage of this is that it uses a lot of memory since every vertex is stored even if many building models share the same geometry/basic shapes but are only at different positions.

Instanced rendering is a method to render shapes that share the same geometry changing only the transformation between instances. OpenGL allows to render instanced geometry by passing the model matrix as input to the shader. The advantage of this is that the geometry for identical shapes only needs to be stored once. Every instance of this shape is rendered using the corresponding model matrix.

3.2 Texture support

Texture support is needed to actually render terminal shapes with different textures and allow to create so called surrogate textures. In the original *PGA* implementation there is no material support and thus is implemented for this work. To render geometry with different textures the rendering buffers are split in so called terminal buffers. Each terminal buffer holds all the geometry that is rendered with a specific texture. This is needed to render the each terminal buffer with a corresponding texture. To render terminal buffers with texture mapping the geometry generating operator also produces texture coordinates for every terminal symbol. The texture mapping is created depending on the used terminal shape.

This also allows to create different materials (texture + shader combinations) for every terminal buffer. During runtime the corresponding shader assigned to this terminal buffer can be used to render the geometry.

The generation of the texture coordinates depends on the shape that is generated. Since the Generate operator is the only one that generates geometry information (vertex position, normals, etc.) this operator is extended to generate texture coordinates accordingly. Since the operator has the information about the generated shape type it is easy to create the correct texture coordinates for each type. These texture coordinates are stored per vertex like the other vertex attributes. See figure 3.2 for a simple example of a textured house created by a shape grammar with *PGA*.

Figure 3.2: Simple textured house generated by *PGA*.

To apply the texture to the rendered terminal buffer the texture is set as active texture while this buffer is rendered. A vertex shader with the new texture coordinates as additional input and a fragment shader with the active texture as sampler can render the resulting shape with the associated texture.

3.3 Surrogate Terminals

Surrogate Terminals are generated terminal symbols that replace a node in the derivation tree and allow to prune the tree. These terminals need to resemble the terminal shapes that would have been generated by the replaced sub-tree. They need to approximate the replaced geometry, as well as the replaced texture detail. For the geometric detail the derived shape can be used to generated a suitable surrogate geometry. For the texture detail it is necessary to use all the textures used by the replaced terminal shapes to create a so called surrogate texture. This texture approximates the replaced detail of the sub-tree.

The advantage of surrogate terminals is that the grammar evaluation can stop at a certain point, when no more detail is needed for the current scene. A lot of geometry that would have been generated can be skipped this way. This results in a reduced time to generate the resulting scene. See figure 3.3 for a example of this.

As you can see the non-terminal symbol FacadeTile is replaced by the termi-

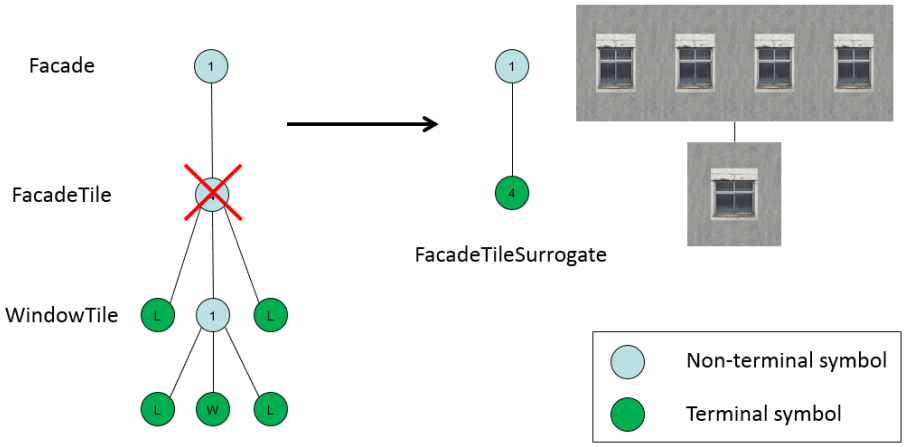


Figure 3.3: Surrogate terminal replaces a sub-tree in the derivation tree.

nal symbol FacadeTileSurrogate. This means that the grammar evaluation stops at this point and the sub-tree of the replaced node is pruned. Since every rule works on a current shape, the shape of the replaced symbol is used as shape for the new terminal symbol. Since the evaluation stops at this point a generate operator is executed on the current shape to create geometry in the current scope. To store the generated geometry it is necessary to provide additional terminal buffers for every surrogate terminal in the grammar. These buffers are used to render the surrogate terminals with a special texture. As the replaced sub-tree can have multiple terminal symbols as leaf nodes, the surrogate terminal needs to consider the detail covered by those nodes.

3.4 Surrogate Textures

Surrogate Textures are the textures used to replace the detail covered by the original sub-tree of the shape grammar derivation. As mentioned before, every surrogate terminal has its own terminal buffer. Now a texture is needed to render these terminal buffers. The shape of the surrogate terminal covers the spatial scope of the replaced rules. Because of that the shape of the replaced rule is used as scope for the surrogate texture as well.

In order to create surrogate textures for candidate nodes the derivation tree for the corresponding grammar has to be parsed bottom up starting with the textures for terminal symbols. To do this a suitable data structure is needed to actually traverse the derivation tree. Since the shape grammar rule-set is known, it can be used to create a tree structure that resembles the derivation tree of this shape grammar. *PGA* uses templates to define the rule-set, a similar method can be used to build a tree structure. Every rule has the next successor rule as a parameter, thus defining the tree structure

implicitly. For each rule a corresponding node in the tree is created. The tree itself is build starting from the first rule as root node, each successor rule is added as a child to the current node. This is done for every rule until the terminal nodes have been reached. They represent the leaf nodes in the tree.

After the tree has been build, a depth first search algorithm can be used to traverse the tree. To start the surrogate creation process the search starts with the root node and traverse the left most branch of the tree until it reaches the first leaf node. At the leaf nodes, the texture that is used for this terminal shape is added possible child texture for intermediate surrogate textures. A list of surrogate terminals is filled while traversing the tree. Depending on the operator type and the shape a surrogate terminal is created using the current shape as geometric scope. The textures from the child nodes are used to compose a texture that represents the child textures. This is done with a render-to-texture method. Since the scope of the parent and child nodes is known from the parameters of the shape grammar the textures can be placed accordingly and then rendered to a new texture. The texture is saved together with the shape of the terminal. This is done for all split operations that use a quad shape (subdivide, repeat). See figure 3.4 for an example surrogate texture created by this process.

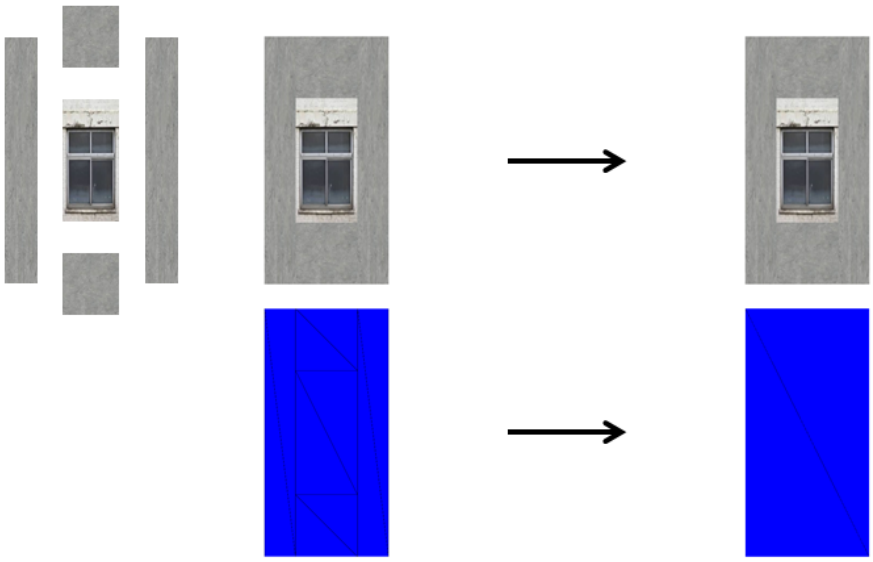


Figure 3.4: Surrogate texture created by render-to-texture

As you can see the created surrogate texture resembles the replaced child nodes. The shape of the surrogate terminal is the current shape of the replaced node. After the textures have been created the are simply stored on disk. This whole process can be done as a preprocessing step for a given shape grammar. During runtime the textures

are loaded and used when rendering the corresponding terminal buffer of the terminal.

3.5 Level of Detail

To apply the created *LOD* textures during runtime it is necessary to use a condition to apply the right detail level. The distance between current terminal and the camera is used as condition to derive the right level. These distances are used during runtime to decide if the current grammar evaluation can halt and a surrogate terminal can be used. These distances can be derived offline during the pre processing step after the surrogate terminal are created.

To do this, the surrogate terminals with their corresponding textures and the original detailed geometry are rendered and compared to each other. The comparison is done using HDR-VDP-2 [10], a visual metric for image comparison. This is done for different camera distances until a error threshold is reached otherwise the distance is set to the far plane.

The original rendering is done in a higher resolution to overcome sampling errors due to rasterization and aliasing. Because of the sampling theorem the minimum sampling frequency has to be bigger than two times the input frequency. In case of an image the sampling is done in two dimensions. This means that the number of pixels needs to be at least four times higher than the resulting resolution. See figure 3.5 for an example of surrogate terminals at different distances.

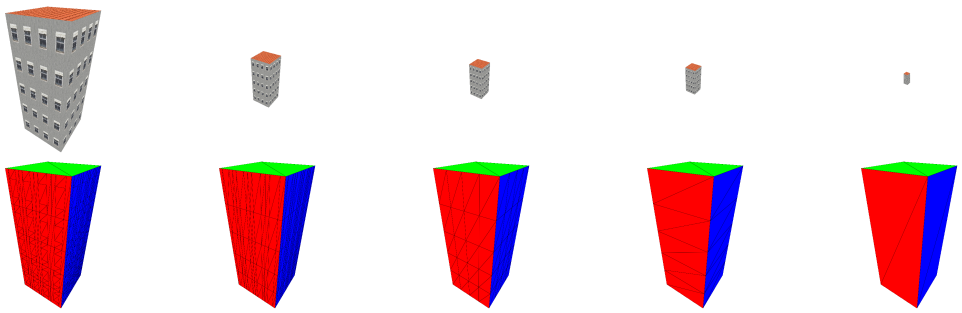


Figure 3.5: Renderings of the building at different detail levels

3.6 Stochasticity

Stochasticity allows to derive many different final output shapes from a single shape grammar. So instead of using several grammars for different building you can use only one grammar that is parametrized with random values to allow different outcomes in the

evaluation phase.

To incorporate random parameters into *PGA* there are two things needed: A random seed and a random generator to propagate the seed. The seed is generated at the beginning as a pseudo random value for every starting symbol (axiom). This seed is stored as an attribute of the shape, so each shape needs an additional memory of four byte.

During the evaluation of the grammar rule the seed is used to generate a new seed for the successor rule. The next seed is calculated with a linear congruential generator. This way, the propagation of the seed down the tree is deterministic. Thus, the same initial seed will result in the same derived random values at every node of the derivation tree.

The linear congruential generator has the form: $x_{n+1} = ax_n + b \mod m$

- x_{n+1} is the next seed value
- x_n is the current seed
- a, b constant factors
- m modulus

When using random parameters with the grammar the generation of the surrogate texture differs from the normal method. Since the resulting texture can look different depending on the propagated seed there needs to be a way to approximate all the possible children textures. One way would be to create all possible result textures and save them. The selection of the right texture can be done in the surrogate operator using the random value.

This is not feasible for many nested rules with random parameters since it can result in a large amount of need textures. One way to save texture memory is to cluster resulting sample textures and create one surrogate texture for all samples assigned to this cluster. In figure 3.6 you can see an example of three clustered sample images using k-means.

To compensate the error between the surrogate texture and the original child textures the calculation of the *LOD* error metric has to be adapted. Instead of one sample per distance the error metric has to be evaluated for each resulting cluster texture. To do this a number of samples of the original rendering is created, spanning over the range of the random value. Every sample rendering is compared to the surrogate rendering of the nearest cluster. The sample with the largest error is taken to derive the minimal distance to the camera for every cluster texture.

Figure 3.6: clustered sampled textures

4 Implementation

The implementation is done in C++/*CUDA*. For testing and comparing of the image quality MATLAB is used. To define the grammar C++ template are used. This gives the advantage of generating the derivation tree during compile time.

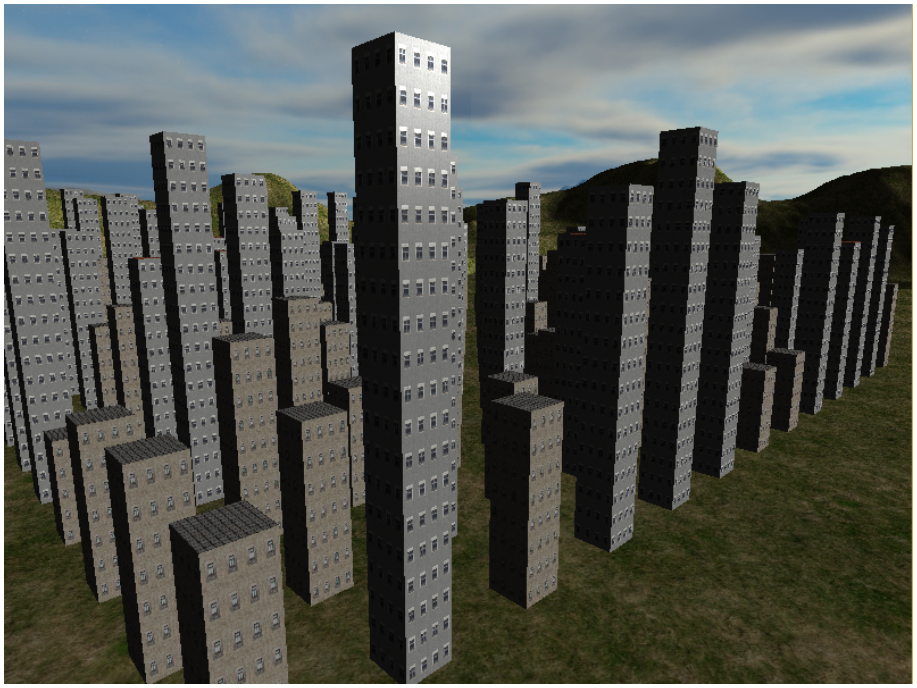


Figure 4.1: Different houses generated with PGA

4.1 PGA extensions

There are two we added to the *PGA* framework to support surrogate terminals. First, texture support is needed to allow the rendering of textured terminal shapes. Second,

random values are needed to allow stochastic evaluation of the shape grammar.

4.1.1 Texture support

To allow rendering the generated geometry with textures, there are two things needed. First, the texture mapping for the terminal symbols has to be defined. And second, the assignment of textures to a specific terminal symbol has to be done. *PGA* is using vertex buffers to store the generated geometry (positions, normals, indices).

The texture mapping depends on the terminal shape created. In case of a quad the UV coordinates can be simply created by mapping the uv plane onto the quad. For cubes a single texture can be used for all six faces.

Operators like repeat apply the same operation multiple times to their parent shape. This can be used to employ the repetition to the texture instead of the geometry. OpenGL allows textures to be repeated across a surface by altering the texture coordinates to control the repetition number. This can be used for surrogate textures to repeat the child texture and approximate the geometry.

4.1.2 Stochasticity

For randomness there are two things needed. First, a randomly seed needs to passed to each rule of the shape grammar. And second, a new rule parameter type which allows to define ranges of possible parameters. The random seed is used to generate the actual value in the valid parameter range which is applied to this operator.

To pass the random seed to all rules in the grammar the derivation tree can be used. Starting with a generated pseudo random value for the start rule every node down the tree uses this value to generate a new number and passes it to its children. This way every node gets a unique random value which is deterministic and dependent on the starting value of the root node. This leads to the same derivation every time the same starting seed has been used.

4.2 Derivation Tree

The grammar derivation tree is implicitly build by the grammar rules using the nested templates that are used to define the grammar. The start rule is the root node, the non-terminal symbols are the intermediate nodes and the terminal symbols are the leaf nodes. To build the tree a node data structure which contains the a reference to the children of the node and information about the used parameters. Since every operator can produce a different number of children a template specialization is used to encode the further derivation and calculation of shape size. The size is needed to derive the

correct texture alignment for the surrogate textures.

To traverse the tree each operator gets a method for building up the related surrogate texture. These methods are called on all child nodes of the current node. This way, the whole tree is traversed in a depth first manner and for each suitable operator a texture can be build. The rule tree is created from nested templates, thus it is build during compile time. All the information needed from the shape grammar is stored inside the template as types. During runtime when the tree is traversed the implemented methods have access to all members of their template class.

4.3 Surrogate Terminals

After the derivation tree has been build, each node contains all the information needed to build surrogate terminals. It depends on the type of operator if a surrogate terminal is created or not. For example a subdivide or repeat rule can be replaced by a fitting surrogate. An operator that is suitable for a surrogate terminal also has a method to create a surrogate texture that resembles the child nodes appearance. Since the derivation tree is traversed from bottom up the surrogate texture creation process starts with the leaf nodes which are assigned to a specific texture/material. These textures are passed to the parent node to create the first surrogate texture where applicable.

The texture is generated based on the appearance of the child nodes. Starting with the leaf nodes the used textures are passed to the parent rules. Depending on the operator and given parameters the child texture are arranged to fit the appearance. This arrangement is rendered to a new texture which is assigned to the current node as surrogate texture. Using the size information previously stored in each node of the derivation tree, the correct alignment of textures can be build. Since this process can be done beforehand as preprocessing step the textures are simple stored for later use.

4.4 Level of Detail

The *LOD* is derived from comparing the generated surrogate texture with the original rendering. To do this an image quality measurement is needed. Both are rendered at different distances and compared given an image quality measure. The distance which reaches a quality threshold is chosen as minimal distance for switching the detail level. The important parameter is the error between original and surrogate rendering since the chosen distances depends on it. The larger the error is, the earlier a non terminal is replaced by an surrogate terminal.

When using random parameters the different resulting outcomes can not be

approximated by one terminal. Instead all the possible outcomes need to be approximated by multiple surrogate textures per terminal. The appearance of child nodes depends on the random seed. To get every possibility, every possible outcome could be rendered to a texture and then chosen during runtime using the corresponding seed value. This can lead to a lot of textures which would need to be generated.

Instead of creating all possible textures similar textures can be clustered to a single texture using a k-means clustering. This way, the amount of textures needed can be greatly reduced. During runtime, the right texture is chosen by selecting the nearest cluster.

4.5 Runtime Integration

To use the surrogate terminals during runtime a new operator is introduced: The Surrogate operator takes as parameters the distance, the buffer index for the surrogate terminal and the rule that is normally derived. This operator simply checks the current distance between camera and the current shape. If the distance is greater than the distance provided by the operator the surrogate terminal is evaluated. Otherwise the grammar derivation simply continues with the next rule.

The Syntax is as follows:

```
Surrogate<maxDistance, CallRule<Successor>, BufferIndex>
```

- maxDistance - minimal distance to the camera
- Successor - next rule to be evaluated
- BufferIndex - index of the terminal buffer

This Operator simple checks $distance(cameraPos, shapePos) < maxDistance$. If the condition is met, the operator creates a terminal symbol within the current scope. The vertex data is stored in the terminal buffer addressed with BufferIndex.

During rendering the surrogate terminals are rendered using the corresponding buffer index defined in the shape grammar. The surrogate textures are assigned to this buffer index, so the terminals are rendered with the correct texture.

There are two ways to incorporate this new operator in a shape grammar. With rule-based scheduling the grammar rules are given as template parameters. For each surrogate terminal a new rule has to be added before the rule that is to be replaced. This rule becomes the successor rule to the surrogate rule in case the condition is fulfilled. When using operator-based scheduling the surrogate rules can be placed in

the dispatch table created by *PGA*. This can be done during runtime since the table is dynamically created in the *GPU* memory.

4.6 Random Parameters

The derived seeds can be used to parametrize the operators with random values. To achieve this, *PGA* needs to distinguish between constant values and random values. Since *PGA* already uses templates to parametrize the rules, new template classes for constant and random values are added.

The random values can be expressed as a closed interval $[a, b]$ so the resulting value x lies within. During runtime the passed seed and the interval result in a pseudo random value as final input parameter for an operator.

Random value calculation: $x = a \cdot seed + |b - a|$

PGA is implemented in C++ using template meta programming for grammar definition and evaluation. This means that the grammar is defined using nested templates classes. The parameters for the rules are also encoded as template arguments. The advantage of this is that you can use different data types as parameters. This means that the grammar definition and evaluation (rule-based evaluation) is done during compile time which saves a lot of processing power during runtime. Syntax for constant and random values:

```
Const<Value>
Rand<Min, Max>
```

Since *PGA* can use two different scheduling methods as mention in the generation section there is a difference in storing these parameters.

With rule-based scheduling the parameters are simply stored as template parameters. Meaning they are set during compile time. While this works for constant values, random values need the seed provided by the shape during runtime. Thus, every operator that support random values needs to be adapted to evaluate the random value during runtime to derive the actual parameter value before executing its operation.

When using operator-based scheduling *PGA* creates a dispatch table for the grammar rules during runtime. This means the parameters also needs to be stored in this table. Each dispatch table entry also holds an array of parameters used for the corresponding operator of this rule. While constant values only need one entry per parameter in this array, random values need more memory since it is necessary to store the minimum and maximum possible value in the table. This means the table needs to

hold at least two times the number of parameters as values.

The seed can not only be used for parameters, there is also a new operator called RandomRule. This operator takes a list of successor rules and a probability value which are used during runtime to determine the successor rule. The syntax for this new rule is as follows:

```
RandomRule<Pair<Propability, CallRule<Successor>>, ...>
```

This rule takes the current seed and selects one of the successor rules based on the probabilities. The probability values are in the range of 0-1 and add up to 1. During runtime, the seed value is used to select one of the corresponding successor rules. The selected rule is evaluated next and the seed is passed to the following rule. This way, different evaluation paths lead to different intermediate symbols and sub-trees which produce a completely new set of geometry symbols. In the context of different starting axioms one can use this operator to produce totally different city buildings based on the random seed. The seed is passed to the children nodes and recalculated per node.

5 Results

This section presents results of the presented approach for surrogate terminals. Specifically, it compares its rendering quality and performance to the traditional approach, i.e. without using surrogate terminals. To judge the presented approach, this section compares the time spend on geometry generation, the overall time spend on rendering one frame (as frames per second) and the amount of geometry built.

The approach is evaluated on three different scenes. The first scene consists of a single building without any random parameters. For the second scene, two different building rule sets are developed and chosen randomly. In addition, the height of all buildings is randomized. The selection of the building rule and the height requires the system to use the random seed to select the right surrogate terminal at runtime. The last scene shows more complex office buildings with random elements.

All tests have been done on a common PC system operating on Windows 10. The system is equipped with an Intel Core i7-4770k CPU, 16GB RAM and a NVIDIA GeForce GTX 980 TI, 6 GB VRAM graphics board.

5.1 Uniform Buildings

A simple rule set is used to test the functionality of the surrogate terminals during runtime. (see Figure 5.1). The entire scene consists of 256 houses placed in a 16 by 16 grid. To get consistent measurements the camera is placed at a fixed location where the whole scene of buildings is visible.

The renderings with and without surrogate terminal are shown in Figure 5.2. Notice the reduced amount of polygons when surrogate terminals are used. Both rendered image look almost the same, it is hardly possible to spot a difference with

the naked eye. But when looking at the wireframe models of the same renderings the difference is much more obvious. In the close up of one of the buildings it is visible that there is basically only one quad per side of the house. In the original scene each side has much more geometry rendered.

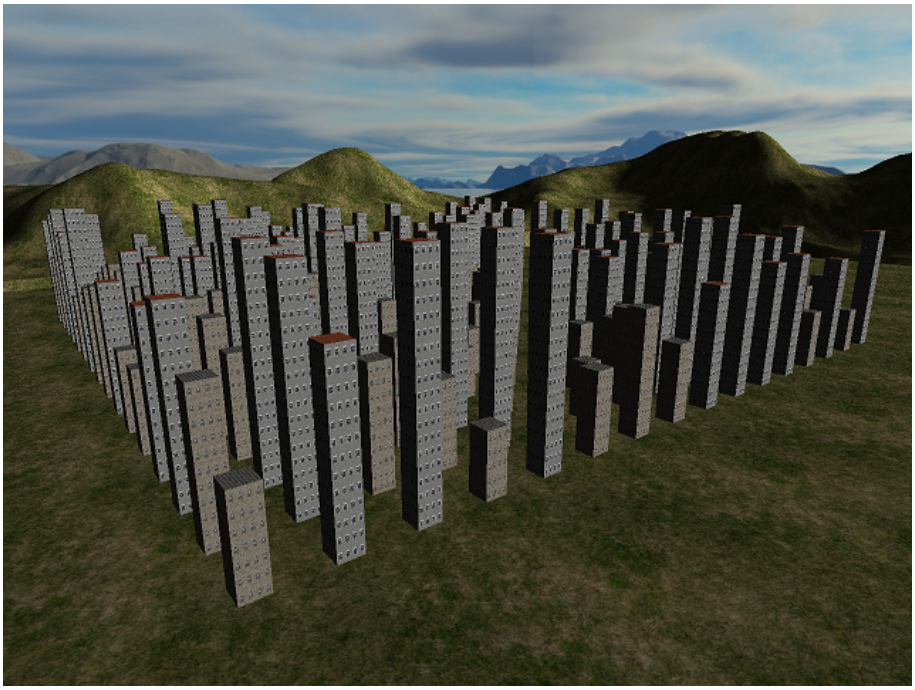


Figure 5.1: Uniform Houses

SimpleHouse	Surrogates	no Surrogates
Average FPS	4143	1925
Generation time	0.087616 ms	1.65578 ms
Vertices	8848	994368
Indices	13272	1491552

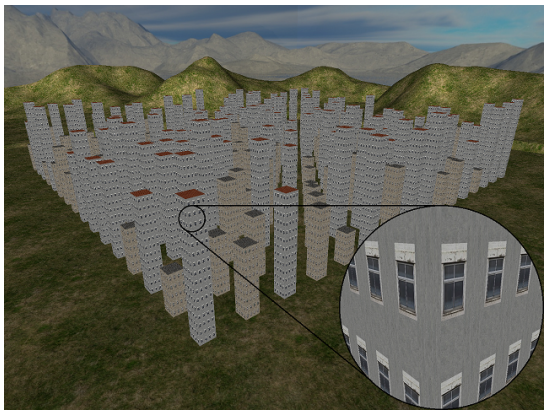
Table 5.1: Results for the Simple House testcase

Table 5.1 shows the average generation time for all the buildings and the average rendering time, measured in frames per second (fps). The number of vertices and indices show the amount of geometry rendered. Since the terminal shapes are quads in this case there are 4 vertices and 6 indices per quad.

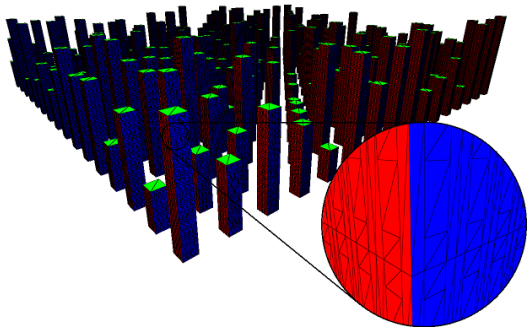
The measurements clearly show the reduction of geometry needed to render the scene compared to the original scene without surrogate terminals. Also the time

```
Wall -> Generate {Quad}
Roof -> Generate {Quad}
Window -> Generate {Quad}
WindowTile -> SubdivideY {1r : Wall, 2r : Window, 1r : Wall}
FacadeTileX -> SubdivideX {1r : Wall, 2r : WindowTile, 1r : Wall}
FacadeX -> RepeatX {0.5 : FacadeTileX}
Side -> RepeatY {1.0 : FacadeX}
First -> ComponentSplit {Roof, Wall, Side}
```

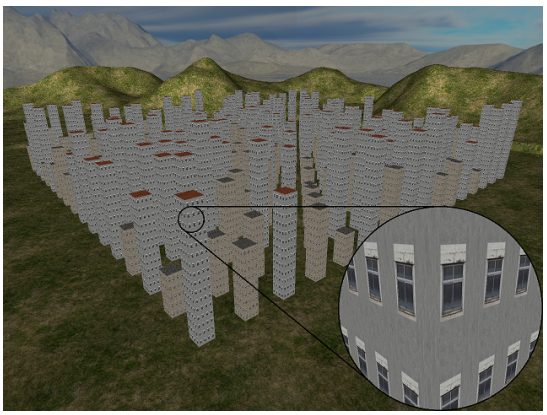
(a) Grammar to generate a textured house.



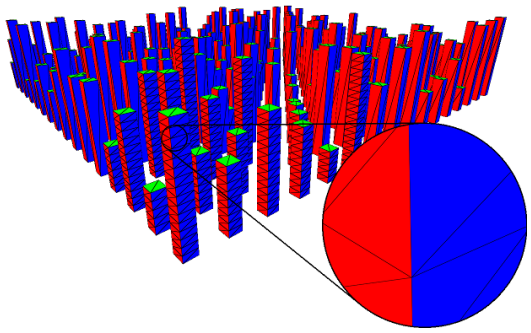
(b) Rendering without surrogate terminals.



(c) Wireframe rendering without surrogate terminals.



(d) Rendering with surrogate terminals.



(e) Wireframe rendering with surrogate terminals

Figure 5.2: Uniform Building Scene. (a) The grammar used to render the scene. (b) Rendering including texture buildings without the surrogate terminal approach. (c) Wireframe rendering of the scene without using surrogate terminals. (d) Rendering with textures using surrogate terminals. (e) Wireframe rendering using surrogate terminals.

spend on generating the geometry is much lower since the evaluation of the shape grammar can stop if the condition for the surrogate operator is met. However, the resulting rendering looks very similar to the one created without using surrogate terminals. The reason for this is that the surrogate textures used to replace the detailed geometry are created from the child node textures. Thus they should closely resemble the replaced detail.

5.2 Random Buildings

In order to test randomness there are two things to consider. First random parameters that allow to alter the derived output shape(s) of a rule. Second the new RandomRule operator that allows to randomly select the next rule within a set of possible successor rules. Both cases are tested with and without surrogate terminals to look at the performance using randomization.

5.2.1 Random Windows

This scene shows the same buildings as in the first testcase with the difference that the amount of windows in each floor can be between 4 to 6. This shows that the surrogate terminal also work for a random repeat operator. The scene consists of 256 houses in total, placed on a 16 by 16 grid (see Figure 5.3).

In this test the same rule-set as in the simple house testcase is used but this time with random parameters. Table 5.2 shows the results of this testcase. As you can see, the amount of windows is the same in both renderings but with surrogate terminals the amount of geometry is much less. Because of the repeating pattern in the geometry it can easily be replaced by a repeated texture.

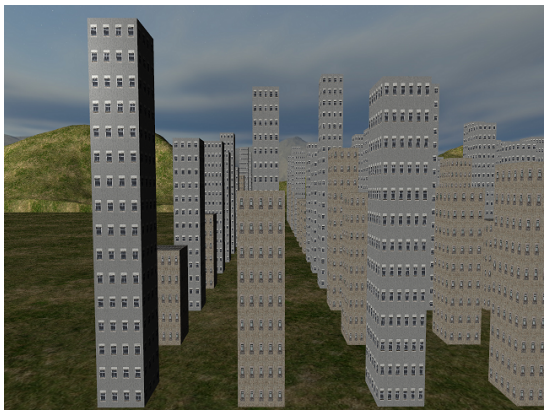
RandomWindow	Surrogates	no Surrogates
Average FPS	4455	1744
Generation time	0.09104 ms	1.98877 ms
Vertices	8848	1166128
Indices	13272	1749192

Table 5.2: Results for the Random Window testcase

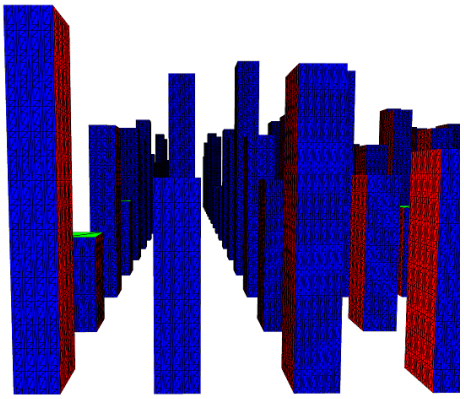
The generation time is much less with surrogate terminals than without, also the amount of vertices and indices needed to be stored for rendering is drastically reduced.

```
Wall -> Generate {Quad}
Roof -> Generate {Quad}
Window -> Generate {Quad}
WindowTile -> SubdivideY {1r : Wall, 2r : Window, 1r : Wall}
FacadeTileX -> SubdivideX {1r : Wall, 2r : WindowTile, 1r : Wall}
FacadeX -> RepeatX {Rand(0.3, 0.5) : FacadeTileX}
Side -> RepeatY {1.0 : FacadeX}
First -> ComponentSplit {Roof, Wall, Side}
```

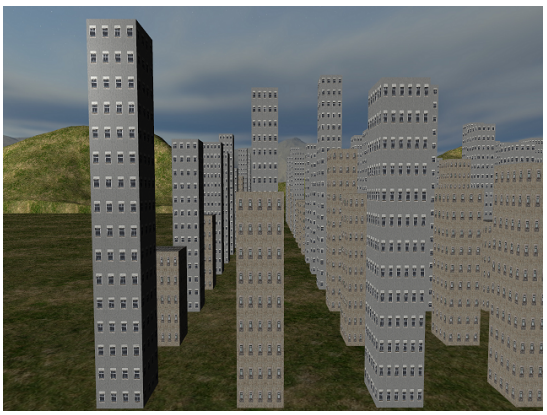
(a) Grammar to generate a textured house.



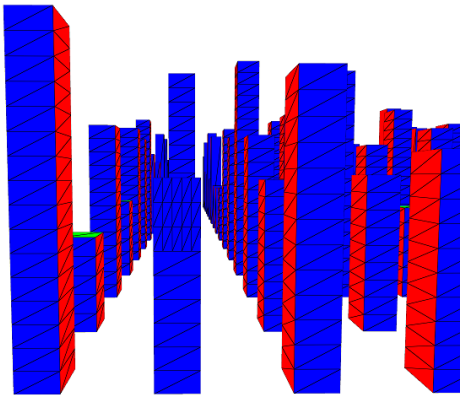
(b) Rendering without surrogate terminals.



(c) Wireframe rendering without surrogate terminals.



(d) Rendering with surrogate terminals.



(e) Wireframe rendering with surrogate terminals

Figure 5.3: Random Window Scene. (a) The grammar used to render the scene, notice that the repeat parameter for the facade is randomized. (b) Rendering of buildings with variable number of windows without surrogate terminals. (c) Wireframe rendering of the scene without using surrogate terminals. (d) Rendering with textures using surrogate terminals. (e) Wireframe rendering using surrogate terminals.

5.2.2 Random Facades

This Testcase provides 5 different facade types for the same House. The RandomRule operator is applied after the base shape of the building is set. This operator selects one of the five facade rulesets depending on a probability and the given random value. See figure 5.4 for the five facades.

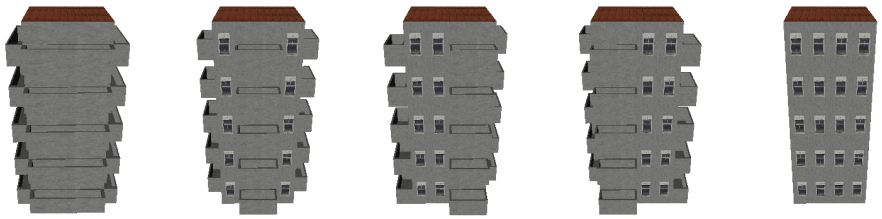


Figure 5.4: Five different facades for the same house

For this testcase a 16x16 grid of skyscrapers is generated and rendered. The generation

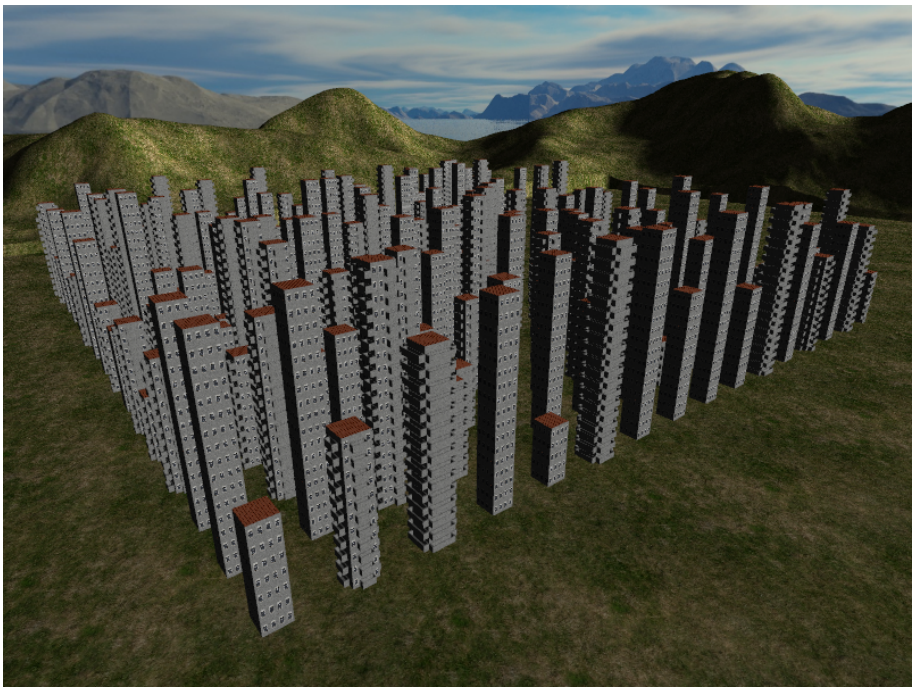


Figure 5.5: Rendering of generated houses different facades

and rendering time is compared with surrogate terminals and without them. To generated the same buildings the same seed is used to derive the random values. Thus the city has exactly the same buildings in both renderings. As you can see the generation time with surrogate terminal is about half of the original scene. Also a lot less vertices

RandomRule	Surrogates	no Surrogates
Average FPS	1274	922
Generation time	1.45ms	2.91ms
Vertices	820480	1420752
Indices	1230720	2131128

Table 5.3: Results for the RandomRule testcase

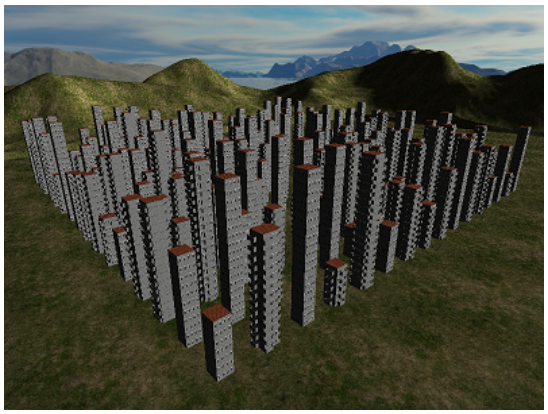
and indices are needed to render the scene resulting in more frames per second. See figure 5.6 for a side by side comparison between both scenes and their wireframe models.

```
Wall -> Generate {Quad}
Roof -> Generate {Quad}
Window -> Generate {Quad}

BalconyWall -> Generate {Box}
BalconyEmpty -> Discard
WindowTile -> SubdivideY {1r : Wall, 2r : Window, 1r : Wall}
FacadeTileX -> SubdivideX {1r : Wall, 2r : WindowTile, 1r : Wall}
BalconyTile -> SubdivideX {1r : BalconyWall, 18r : BalconySpace, 1r : BalconyWall}
BalconyOutside -> SubdivideZ {4r : BalconyTile, 1r : BalconyWall}
BalconyExtent -> SubdivideY {1r : BalconyEmpty, 1r : BalconyOutside}
Balcony -> Extrude {0.5 : BalconyExtent}

FacadeA -> RepeatX {0.5 : FacadeTileX}
FacadeB -> SubdivideY {1r : Balcony, 1r : Wall}
FacadeC -> SubdivideX {1r : FacadeTileX, 2r : FacadeB, 1r : FacadeTileX}
FacadeD -> SubdivideX {1r : FacadeTileX, 1r : FacadeTileX, 2r : FacadeB}
FacadeE -> SubdivideX {2r : FacadeB, 1r : FacadeTileX, 1r : FacadeTileX}
FacadeX -> RandomRule {FacadeA, FacadeB, FacadeC, FacadeD, FacadeE}

Side -> RepeatY {1.0 : FacadeX}
First -> ComponentSplit {Roof, Wall, Side}
```

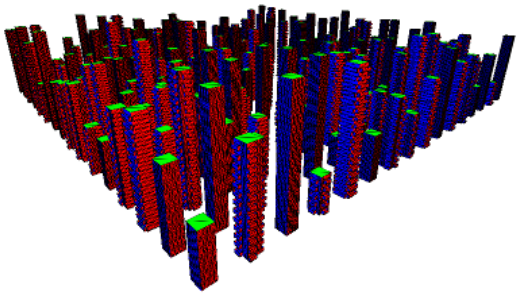
The above Listing shows the shape grammar used to generate the different façades. Note the RandomRule operator that selects one of the five possible façades. Each façade has a value assigned which determines the probability of being chosen as next rule. These probabilities always add up to 1.

```
BalconyWall -> Generate {Box}
BalconyEmpty -> Discard
BalconyTile -> SubdivideX {1r : BalconyWall, 18r : BalconySpace, 1r : BalconyWall}
BalconyOutside -> SubdivideZ {4r : BalconyTile, 1r : BalconyWall}
BalconyExtent -> SubdivideY {1r : BalconyEmpty, 1r : BalconyOutside}
Balcony -> Extrude {0.5 : BalconyExtent}
```

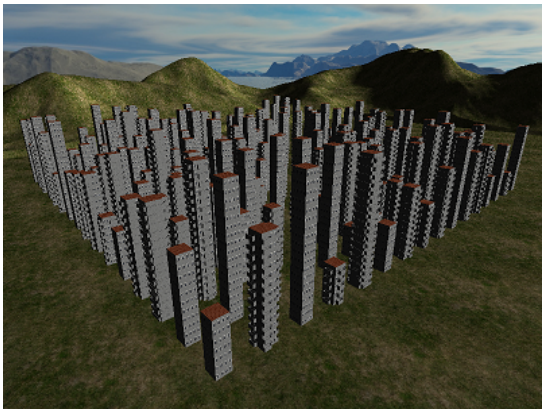
(a) Grammar to generate a balcony.



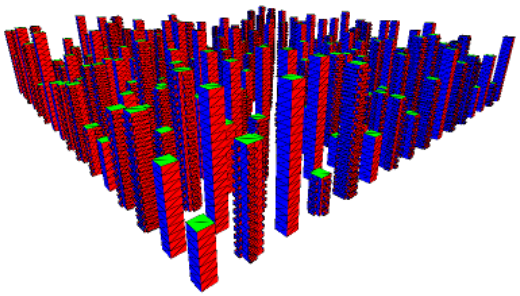
(b) Rendering without surrogate terminals.



(c) Wireframe rendering without surrogate terminals.



(d) Rendering with surrogate terminals.



(e) Wireframe rendering with surrogate terminals

Figure 5.6: Random Rule Scene. (a) The grammar used to render the scene, notice that the repeat parameter for the facade is randomized. (b) Rendering of buildings with variable number of windows without surrogate terminals. (c) Wireframe rendering of the scene without using surrogate terminals. (d) Rendering with textures using surrogate terminals. (e) Wireframe rendering using surrogate terminals.

5.3 Complex Buildings

For the last testcase a more complex rule-set is used to create office buildings. In this case the façades are split up in a ground façade consisting of the door and a ground floor. The rest of the façade is randomly chosen from three possible alignments. The roof can hold addition decoration with air conditioners and towers.

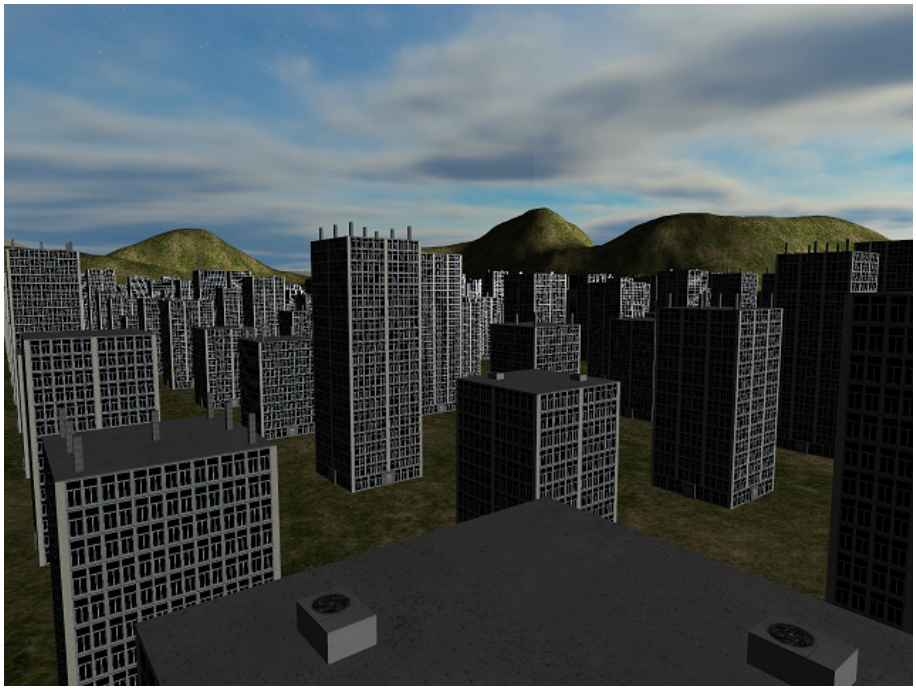


Figure 5.7: Five different facades for the same house

Table 5.4 shows the result of this testcase. Once again the generation time is faster with surrogate terminals than without. The amount of geometry needed is greatly reduced.

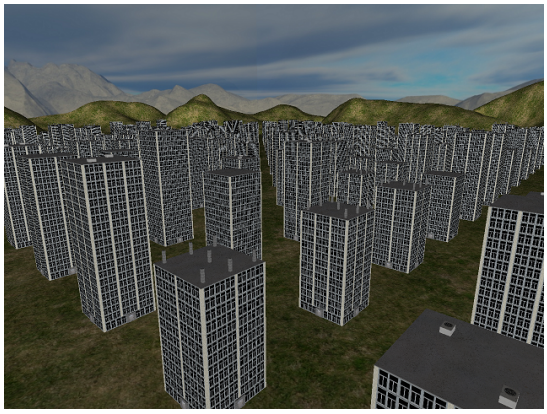
ComplexHouse	Surrogates	no Surrogates
Average FPS	2587	2150
Generation time	0.38576 ms	1.14048ms
Vertices	119116	524288
Indices	178674	786432

Table 5.4: Results for the Complex house testcase

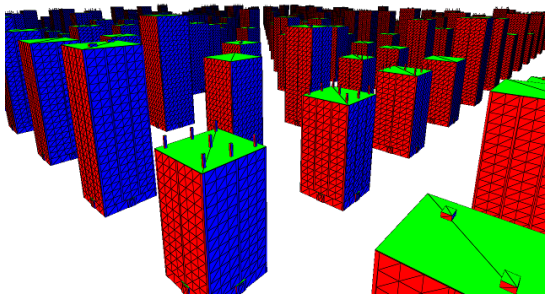
```
AirConditionerSide -> Generate{Quad}
AirConditionerTop -> Generate{Quad}
ColumnWall -> Generate{Quad}
```

```
Door -> Generate{Quad}
Top -> Generate{Quad}
Wall -> Generate{Quad}
Window -> Generate{Quad}

Tower -> ComponentSplit{Wall, Empty, Wall}
Tower4 -> Translate{(0, 2.5, 0) : Tower}
Tower3 -> Scale{(1, 5, 1) : Tower4}
Tower2 -> SubdivideZ{1r : Tower3, 1r : Empty, 1r : Tower3, 1r : Empty, 1r : Tower3}
Tower1 -> SubdivideZ{1r : Tower3, 3r : Empty, 1r : Tower3}
TowerDeco -> SubdivideX{1r : Tower1, 1r : Empty, 1r : Tower2, 1r : Empty, 1r : Tower1}

AirConditioner -> ComponentSplit{AirConditionerTop, Empty, AirConditionerSide}
AirConditioner4 -> Translate{(0, 0.75, 0) : AirConditioner}
AirConditioner3 -> Scale{(3.5, 1.5, 2.5) : AirConditioner4}
AirConditioner2 -> SubdivideZ{1r : Empty, 1r : AirConditioner3}
AirConditioner1 -> SubdivideZ{1r : AirConditioner3, 1r : Empty}
AirConditionerDeco -> SubdivideX{1r : AirConditioner1, 1r : AirConditioner2}

RoofDeco -> RandomRule{AirConditionerDeco, Empty, TowerDeco}
RoofFacade -> ComponentSplit{Top, Empty, Wall}
RoofFloor -> SubdivideY{1r : RoofFacade, 1r : RoofDeco}
RoofExtent -> SubdivideY{1r : Empty, 1r : RoofFloor}

Floor -> RepeatX{4 : Window}
FacadeTile -> RepeatY {4 : Floor}
FacadeLong -> SubdivideX{1 : ColumnWall, 1r : FacadeTile, 1 : ColumnWall,
1r : FacadeTile, 1 : ColumnWall, 1r : FacadeTile, 1 : ColumnWall}
FacadeMedium -> SubdivideX{1 : ColumnWall, 1r : FacadeTile, 1 : ColumnWall,
1r : FacadeTile, 1 : ColumnWall}
FacadeShort -> SubdivideX{1 : ColumnWall, 1r : FacadeTile, 1 : ColumnWall}
Facade -> RandomRule{FacadeShort, FacadeMedium, FacadeLong}

DoorFrame -> SubdivideX {2r : Wall, 78r : Door, 20r : Wall}
DoorFront -> SubdivideX {1r : Wall, 2.4 : DoorFrame, 1r : Wall}
DoorExtend -> ComponentSplit {DoorFront, Empty, Wall}
DoorSection -> ExtrudeZ {1 : DoorExtend}
GroundFacade -> SubdivideX{1 : ColumnWall, 1r : FacadeTile, 5 : DoorSection,
1r : FacadeTile, 1 : ColumnWall}

Roof -> ExtrudeZ {0.8 : RoofExtend}
```

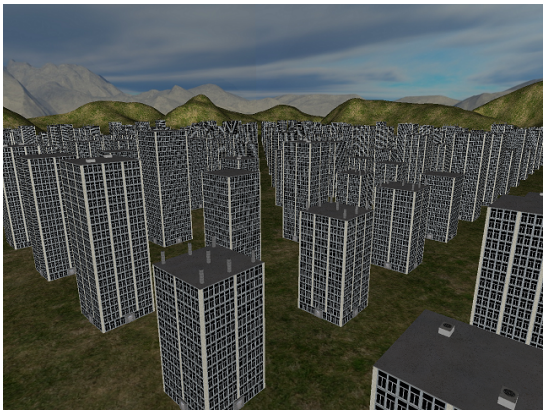
```
Side -> SubdivideY {5 : GroundFacade, 1r : Facade}
Start -> ComponentSplit {Roof, Empty, Side}
```



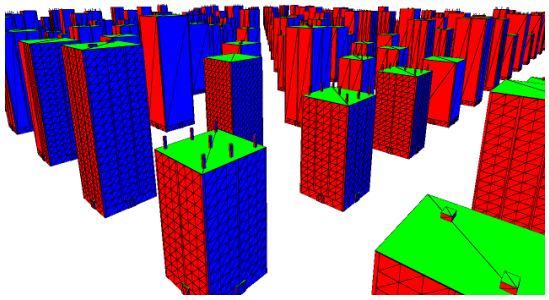
(a) Rendering without surrogate terminals.



(b) Wireframe rendering without surrogate terminals.



(c) Rendering with surrogate terminals.



(d) Wireframe rendering with surrogate terminals

Figure 5.8: Complex house scene. (a) Rendering of buildings with variable number of windows without surrogate terminals. (b) Wireframe rendering of the scene without using surrogate terminals. (c) Rendering with textures using surrogate terminals. (d) Wireframe rendering using surrogate terminals.

5.4 Rendering Quality

The HDR-VDP compares the visibility aswell as the image quality between two images. To test the image quality between the original rendering and the surrogate terminal, the original is rendered at eight times the screen resolution and subsampled. The rendered image is scaled down to original screen resolution using nearest neighbour and bilinear

interpolation. The metric is evaluated between those two scaled down images to get the image quality of the normal rengering. To get the quality of the surrogate rendering the metric is evaluated between the scaled down original renderering and the surrogate rendering.

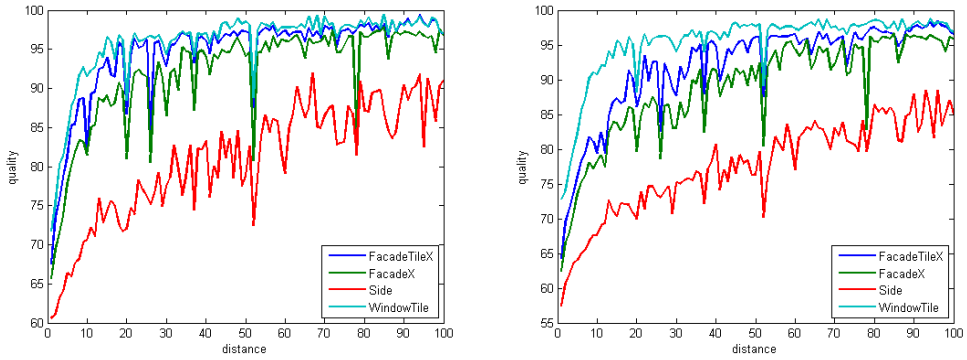
The tests are done using the surrogate terminals created for the uniform house testcase. The each surrogate terminal and the corresponding detailed geometry are both rendered to a texture for image comparison (See Figure 5.9).

(a) Façade side with detailed geometry **(b)** Façede side with a surrogate terminal

Figure 5.9: Examples for rendered textures which are compared. (a) Shows a detailed rendering of one side. (b) Shows the corresponding surrogate terminal with its surrogate texture.

On small distances the quality of the original image is much better compared to the quality of the surrogate. As the distance increases the difference between both qualities decreases. If the difference is below a certain threshold the distance is chosen as switch distance for this surrogate terminal.

As you can see the quality of of both renderings is not much different at low distances. At higher distances the surrogate rendering has a better quality than the original rendering. This is due to geometric aliasing. The difference between both quality measures increases as the distance increases. A negative value indicates that the surrogate rendering has a better quality than the original rendering.



(a) Quality with linear filtering **(b)** Quality with mipmapping

Figure 5.10: HDR-VDP-2 Quality between original and surrogate rendering over different distances. (a) Shows the quality of all terminals using linear texture filtering. (b) Shows the quality using mipmapping.

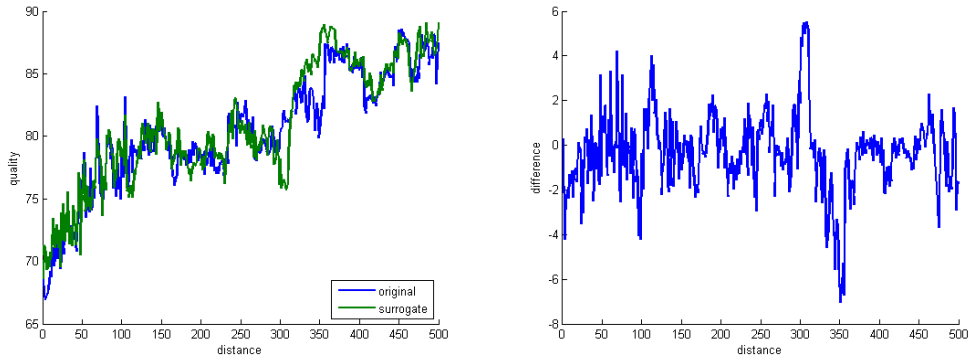


Figure 5.11: Quality of original rendering and surrogate rendering

5.5 Rendering Quality with Stochacitity

With random paramaters the outcome of the generation process can differ a lot. As discussed in chapter 3 a clustering approach is used to reduce the amount of needed surrogate textures in the case of random parameters. The k-means algorithm is used to cluster a number of samples over the range of the random values to k clustered textures. To measure the quality of the clustered texture compared to the original rendering the quality of every sample is compared to the quality of its nearest cluster. See figure 5.12 for an example of this.

Four samples are clustered to the same texture. The quality of the surrogate roughly follows the quality of the original samples.

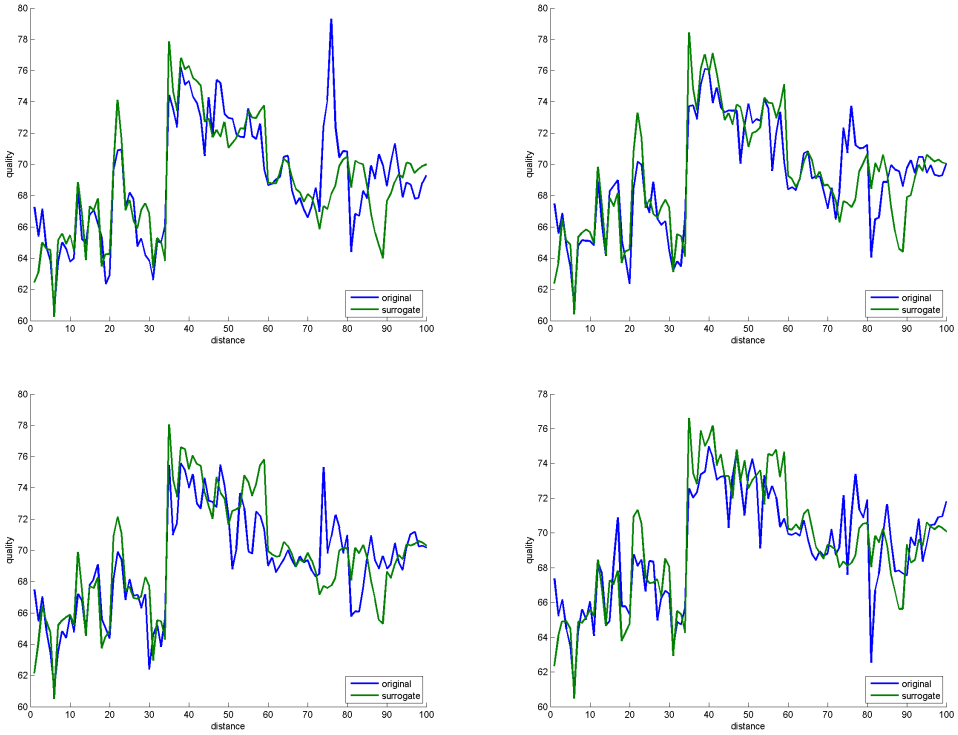


Figure 5.12: Quality of original and surrogate rendering with clustered textures

6 Conclution and Future Work

6.1 Conclustion

Surrogate terminals offer a way to automatically reduce the complexity of generated geometry from shape grammars. The derived surrogate textures replace the covered detail in the grammar derivation tree. These textures can be created as a preprocessing step for a given grammar and applied during runtime via an operator. The needed LODs are derived using a error metric based on visibility and quality of two images.

With these resulting textures the grammar evaluation for a shape can be stopped if the camera distance is large enough that the difference between the surrogate and the full detail cannot be seen. This way the generation time is a lot less compared to evaluating the whole grammar every time. Also the amount of needed geometry is greatly reduced which results in a increased performance while rendering.

6.2 Future Work

In this work the focus was to give an overview what surrogate terminals are and how to derive them for split rules. It would be interresting to extend this to other operators. The problem with operators which change the shape in a three dimensional way is that a surrogate would look quite different depending on the viewing angle. For example a balcony that is created with the extrude operator. Another problem occurs if the extruded shape changes the silhouette of the parent shape.

Solutions would be either to create multiple surrogate textures per terminal that are applied depending on the current viewing angle to the camera. These texture could be clustered like the surrogate texture samples with random parameters.

A

List of Acronyms

API	Application Programming Interface
CGA	Computer Generated Architecture
CGI	Computer Generated Imagery
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GML	Generative Modelling Language
GPU	Graphics Processing Unit
LOD	Level of Detail
PGA	Parallel Generation of Architecture

Bibliography

[1] Besuievsky, G. and Patow, G. (2013). Customizable LoD for procedural architecture. *Comp. Graph. Forum*, 32. (page 15)

[2] Havemann, S. (2005). *Generative Mesh Modeling*. PhD thesis, TU Braunschweig. (page xiii, 7, 11)

[3] Jeschke, S., Wimmer, M., and Purgathofer, W. (2005). Imagebased representations for accelerated rendering of complex scenes. *Eurographics*, pages 1–20. (page 15)

[4] Krecklau, L., Pavic, D., and Kobbelt, L. (2011). Generalized Use of Non-Terminal Symbols for Procedural Modeling. 29:2291–2303. (page 11)

[5] Lefebvre, S., Hornus, S., and Lasram, A. (2010). By-example synthesis of architectural textures. *ACM Trans. Graph.*, 29:A84. (page xiii, 11, 12)

[6] Lin, J., Cohen-Or, D., Zhang, H., Liang, C., Sharf, A., Deussen, O., and Chen, B. (2011). Structure-Preserving Retargeting of Irregular 3D Architecture. *ACM Trans. Graph.*, 30(6):A183. (page xiii, 12, 13)

[7] Lipp, M., Wonka, P., and Wimmer, M. (2010). Parallel Generation of Multiple L-systems. *Computers & Graphics*, 34(5):585–593. (page 9, 13, 14)

[8] Luebke, D., Watson, B., Cohen, J. D., Reddy, M., and Varshney, A. (2002). *Level of Detail for 3D Graphics*. Elsevier Science Inc., New York, NY, USA. (page 14)

[9] Magdics, M. (2009). Real-time Generation of L-system Scene Models for Rendering and Interaction. In *Spring Conf. on Computer Graphics*, pages 77–84. Comenius Univ. (page 14)

[10] Mantiuk, R., Kim, K. J., Rempel, A. G., and Heidrich, W. (2011). Hdr-vdp-2: A calibrated visual metric for visibility and quality predictions in all luminance conditions. *ACM Transactions on Graphics (Proc. of SIGGRAPH'11)*, 30(4)(40). (page 15, 24)

[11] Merrell, P. and Manocha, D. (2011). Model Synthesis: A General Procedural Modeling Algorithm. *IEEE Trans. Visualization and Computer Graphics*, 17:715–728. (page xiii, 12)

[12] Müller, P., Wonka, P., Haegler, S., Ulmer, A., and Gool, L. V. (2006). Procedural Modeling of Buildings. *ACM Trans. Graph.*, 25(3):614–623. (page 7, 10, 11, 13)

[13] Musgrave, F. K., Kolb, C. E., and Mace, R. S. (1989). The synthesis and rendering of eroded fractal terrains. *SIGGRAPH Comput. Graph.*, 23(3):41–50. (page 8)

[14] Perlin, K. (1985). An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296. (page 8)

[15] Prusinkiewicz, P. and Lindenmayer, A. (1990). *The Algorithmic Beauty of Plants*. New York. (page 9, 11)

[16] Schroeder, W. J., Zarge, J. A., and Lorensen, W. E. (1992). Decimation of triangle meshes. *SIGGRAPH Comput. Graph.*, 26(2):65–70. (page 14)

[17] Steinberger, M., Kainz, B., Kerbl, B., Hauswiesner, S., Kenzel, M., and Schmalstieg, D. (2012). Softshell: Dynamic Scheduling on GPUs. *ACM Trans. Graph.*, 31(6):A161. (page 15)

[18] Steinberger, M., Kenzel, M., Kainz, B., Müller, J., Wonka, P., and Schmalstieg, D. (2014a). On-the-fly generation and rendering of infinite cities on the GPU. *Comp. Graph. Forum*, 33. (page xiv, 4, 15, 16)

[19] Steinberger, M., Kenzel, M., Kainz, B., Müller, J., Wonka, P., and Schmalstieg, D. (2014b). Parallel generation of architecture on the GPU. *Comp. Graph. Forum*, 33. (page xiii, 7, 9, 15)

[20] Stiny, G. (1975). *Pictorial and Formal Aspects of Shape and Shape Grammars*. Birkhauser Verlag, Basel. (page 9, 13)

[21] Stiny, G. (1980). Introduction to shape and shape grammars. *Environment and planning B*, 7(3). (page 9)

[22] Stiny, G. (1982). Spatial Relations and Grammars. *Environment and Planning B*, 9:313–314. (page 9)

[23] Wonka, P., Wimmer, M., Sillion, F. X., and Ribarsky, W. (2003). Instant Architecture. *ACM Trans. Graph.*, 22(3):669–677. (page xiii, 4, 10, 17)

[24] Yang, T., Huang, Z., Lin, X., Chen, J., and Ni, J. (2007). A Parallel Algorithm for Binary-Tree-Based String Rewriting in L-system. In *Proc. of the Second International Multi-symposiums of Computer and Computational Sciences*, pages 245–252. (page 14)