Fast Visualization of Large Architectural Models

Artur José Brás Mayer Alkaim

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. António Paulo Teles de Menezes Correia Leitão

Examination Committee

Chairperson: Prof. João Emílio Segurado Pavão Martins
Supervisor: Prof. António Paulo Teles de Menezes Correia Leitão
Member of the Committee: Prof. João António Madeiras Pereira

June 2016

Acknowledgments

I would like to thank my dissertation supervisor Prof. Antonio Leitão for his insight, support and sharing of knowledge that has made this Thesis possible.

I would like to thank my parents and sister for their support, encouragement and caring over all these years, for always being there for me and without whom this work would not be possible.

A special thanks to my girlfriend for the support and patience, which largely helped me making sure I was always motivated, focused and happy.

I would also like to thank my colleagues from the Architecture and Computing group, that gave me critical feedback and one way or another always made me feel accompanied.

Last but not least, to every member of my family and friends that have been with me all these years and helped me to become the person I am today.

To each and every one of you – Thank you.

Abstract

Architects and designers often use Computer Aided Design (CAD) tools to create their models. These tools are very powerful but they are mostly intended for manual use. Unfortunately, the manual production of large amounts geometry is very time consuming.

Generative Design is one approach which considerably speeds up their modeling process. This approach consists in an algorithmic construction of forms and allows the quick creation of massive amounts of geometry. Most CAD tools recognize the usefulness of generative design and provide programming environments for that purpose. However, CAD tools were not originally designed for this type of usage, and favor instead manual use. As a result, they do not have the necessary performance for the effective exploration of this approach.

This work proposes a solution to this performance problem. Through the use of different techniques that speedup the production and visualization of large volumes of geometry, we propose a library designed to support procedural generation. Together with that, we propose a Wrapper API to simplify the library usage that is developed in using the Racket Programming Language. We use it to evaluate the proposal and explore the use of Racket for the generation of large volumes of geometry through Generative Design.

Keywords: 3D modeling, OpenGL, Generative Design, Shaders, Level of Detail

Resumo

Os arquitectos e designers, usam ferramentas de Desenho Assistido por Computador (DAC) para criar seus modelos. Esta ferramentas são muito poderosas para modelação e manipulação dos modelos, mas são desenvolvidas para o uso manual. Infelizmente, a produção manual de grandes quantidades de geometria consome muito tempo.

Desenho Generativo é uma abordagem que acelera consideravelmente o processo de modelação. Esta abordagem consiste em uma construção algorítmica de formas e permite a rápida criação de enormes quantidades de geometria. A maioria das ferramentas de DAC reconhecem a utilidade do desenho generativo e fornecem ambientes de programação para esse fim. No entanto, as ferramentas de DAC não foram originalmente concebidos para este tipo de uso e favorecem o uso manual. Como resultado, eles não têm o desempenho necessário para a exploração efectiva desta abordagem.

Este trabalho propõe soluções para este problema de desempenho, através da utilização de diferentes técnicas que aceleram a produção e visualização de grandes volumes de geometria. É uma biblioteca que implementa várias técnicas e fornece uma API de Modelação 3D com uma interface Racket que acessa esta biblioteca.

Palavras-Chave: Modelação 3D, OpenGL, Design Generativo, Shaders, Nível de Detalhe

Contents

Acronyms

List of Tables

List of Figures

Acronyms

API Application Programming Interface. 6, 29, 38, 56

BIM Building Information Modeling. 31, 42

CAD Computer Aided Design. iii, 3, 5, 29–31, 49, 51

Culling Culling. 10

FPS Frames per Second. 49

GD Generative Design. 3–6, 29, 31, 38

GPU Graphics Processing Unit. 40, 41, 56

HD High-Definition. 23

L-System Lindenmayer System. 16

LOD Level of Detail. xiii, 11, 39, 41

OS Operative System. 55, 56

PL Programming Language. 6, 29, 32, 38, 47, 55, 56

REPL Read Eval Print Loop. 33

VR Virtual Reality. 23

Chapter 1

Introduction

1.1 Motivation

As technology evolves people have more powerful devices and they want to take advantage of that. They want to have more realistic experiences with larger, more detailed and complex contents. This is specially observable in graphic contents. With the recent extra high definition screens and the increasing computational power of the machines, the graphic content has to follow up in quantity as well as in quality.

Graphic content is mainly used for entertainment, both in the gaming and movie industries, but is also used in many other different areas. The fields of architecture and design, for instance, use this technology to experiment and model new designs, from small objects like a plate, to buildings or even entire cities. Unfortunately, manual modeling of large sets of potentially complex shapes is tiresome and very costly.

Figure 1.1 shows a wall that has a relatively large amount of objects (bricks). Here each brick has to be modeled one by one and positioned in its place. This is a rather complex work because each brick has a different position and rotation. This example shows that from a set of simple objects, such as bricks, a complex model can be produced, and the manual modeling process becomes error prone, and time consuming. This problem is even more serious if during the creation steps, architects want to experiment several variations of their designs. In this example of the wall, it is plausible that they want to try different variations of the sine function that defines its curvature.

The simplest solution to this problem is to hire more architects or designers in order to increase productivity and reduce the time needed. However, experience has shown that this solution is not scalable, i.e. doubling the number of architects or designers working in a project will not double their overall productivity. Also, this solution has a big impact on financial costs, that would take immediately out of the market producers with fewer resources.

A solution for this problem is the use of Generative Design (GD). It is a design method that is based on a programming approach in which the users algorithmically describe their models. GD allows architects and designers to model large volumes of complex shapes with significantly less effort. They can model cities, buildings, trees, and many other objects that are, usually, too big or complex for a manual approach. Since the models are represented by a computer program, large volumes of geometry can be generated within short periods of time. This is positive to the users due to the fact that it is more efficient than the manual approach, but, on the other hand, it imposes that users learn how to program to express their models.

Although most CAD applications provide programming languages for generative design, programs

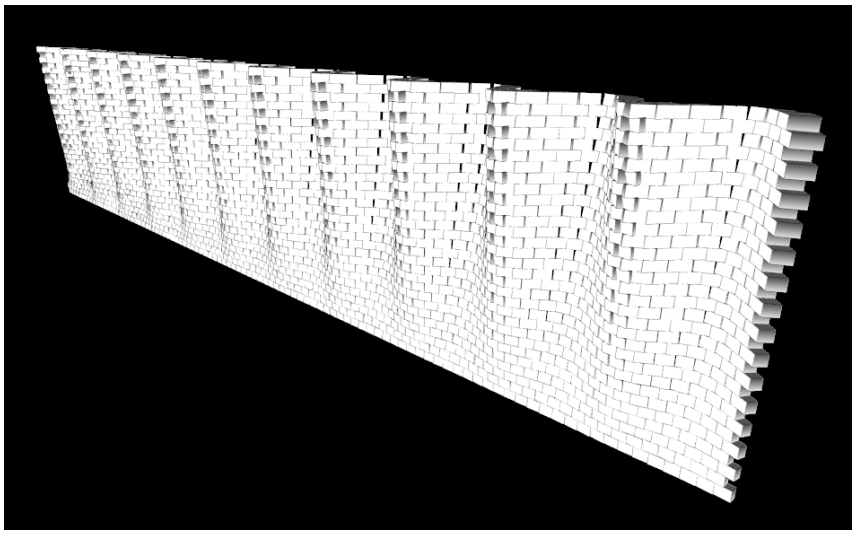

Figure 1.1: Wall of bricks

written in these languages have very limited portability. Additionally, the provided languages, such as AutoLisp [6], C++ or Visual Basic, are not pedagogical and are difficult to use. All these problems create barriers to the adherence to this approach to all users, but specially those that are not used to program [7].

There are several GD tools such as Grasshopper[1] and Rosetta [8], that aim to break down some of these barriers, and facilitate the approximation of these individuals to programming. With these tools, users can create their models using pedagogical and easy-to-use languages.

Typically, these systems implement a straightforward pipeline presented in Figure 1.2.

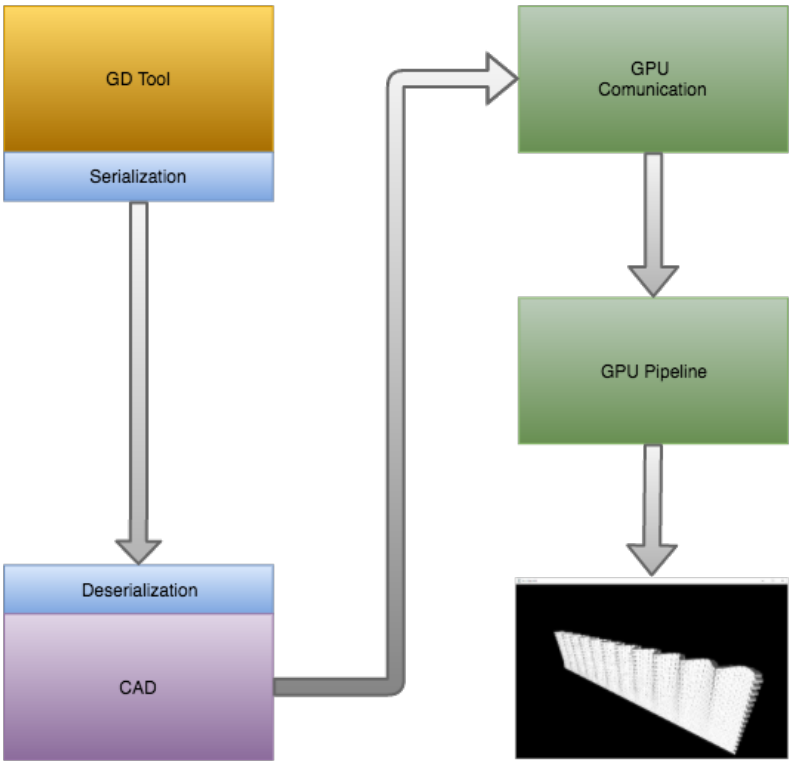

Figure 1.2: Common Generative Design Pipeline

[1] http://www.grasshopper3d.com/

Users implement their models through the GD tool interface. Then all geometry data is serialized and transfered through some transport mechanism. This data has to be deserialized on the other side within the CAD application. The CAD application takes the data and processes it producing geometry. Finally, the geometry is moved to the GPU that renders it. All these steps are time-consuming, due to the large amount of data that needs to be transfered and processed within each step. Adding to this, there is another problem: CAD applications are built for manual modeling mainly, and are not prepared to quickly handle large amounts of geometry. This often causes the tool to freeze, while processing the inputs and sometimes even leads to total crash of the CAD tools.

One big difference between GD and traditional approaches is that users do not see the result of their program while they code. They follow a code-execute-visualize loop where they make changes in the code, execute the code and visualize the resulting model. This makes it difficult for them to understand the impact of changes in their programs, particularly when they have to wait a considerable amount of time to visualize it. It would be much more productive if they could easily understand the correlation between their program and the resulting model and to be able to experiment different values on their program and see the effects they have on the model. To help them with this issue, there is the concept of *immediate feedback*. Immediate feedback is a mechanism that allows the users to quickly see the results of the changes they make. This can be implemented, for instance, through the use of sliders that can be associated with values on the program, and when one slider is moved the effects of that change should be visualized immediately. Grasshopper[2] gained a large user base, not only for being a visual language where the users easily start creating, but also due to the fact that it uses immediate feedback through the use of sliders. Unfortunately, for complex models, the feedback becomes less and less immediate until it just stops.

GD produces much more geometry and at a much faster rate than manual modeling, so the user is able to create massive amounts of geometry, which is fed to the CAD that can become overloaded. With these issues, it is hard to get good performance, specially with large models, which makes impossible to have true immediate feedback. For example, there are a lot of techniques that architects and designers may want to use, such as Fractals (Section 2.5.1), Cellular Automata (Section 2.5.2), and L-Systems (Section 2.5.3), that can generate large amounts of geometry from simple sets of instructions. The use of these techniques is not possible with manual use, due to the large amount of geometry, and is also not possible with the normal pipeline because its performance can not handle the massive amount of geometry that is generated.

This work proposes a solution to this problem and aims to generate large volumes of geometry that is as close as possible to real-time. It does so by jumping over some steps while drastically decreasing the amount of data that is transfered between steps. First we aim to get the geometry as fast as possible to the GPU, so since our goal is just visualization, we jump the CAD layer, eliminating the first communication steps. Another action is to reduce the amount of data that is transferred, by transferring only a very concise description of the geometry, generating the actual geometry on the GPU. To improve visualization performance, techniques such as Level Of Detail (Section 2.4.1) and Culling (Section 2.4.2) are also explored.

[2] `http://www.grasshopper3d.com/`

1.2 Objectives

With the use of GD, architects an designers are able to create very complex models with the composition of large amounts of simple objects, Figure 1.3 is a good example of that. This is a very complex model that is generated from a relatively simple description in Racket. Due to the large amount of geometry, this example is not easy to render with regular GD tools and it takes a lot of time to see the final its rendering.

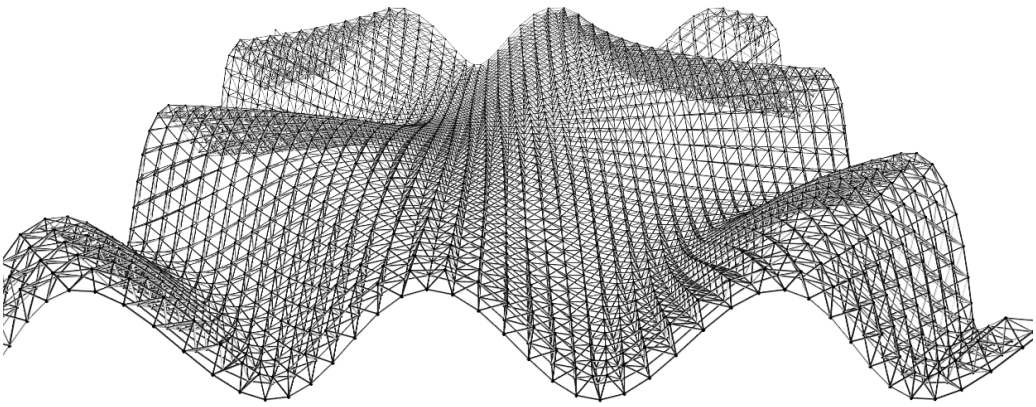

Figure 1.3: Truss Model

The overall goal of our work is to build a GD tool that is able to handle the kind of projects mentioned before. Thus, it should fulfill the following requirements:

- **Performance** - GLFast should be able to quickly generate large amounts of geometry. It should also be able to support the implementation of Immediate Feedback mechanisms to allow the users to quickly see the results of the changes they make, even with large models.
- **Portability** - We want GLFast to be portable and run in the most used systems. It must be implemented so that various Programming Languages (PLs) are able to connect to its Application Programming Interface (API).
- **Completeness** - This work should support a broad set of geometric primitives to provide high creativity freedom to the users.

This work is being developed in the context of the Rosetta Project[3] that is also a GD tool that helps architects and designers to develop their work using GD. Rosetta is an extensible IDE based on **DrRacket** and built in Racket. To allow the connection with Rosetta, the development of a wrapper API in Racket is a complementary objective for this work. In this context the module will act as a fast preview mode that allows the users to rapidly see the changes they make on their model during their creative process.

1.3 Document Structure

This document is structured in the following manner. The next chapter (Chapter 2) will explain several concepts that are used throughout the document, followed by the Related Work (Chapter 3) were we present a set of tools that are related with our work. The Chapter 4 presents our system, first presenting

[3] http://web.ist.utl.pt/antonio.menezes.leitao/Rosetta/index.html

its architecture which is followed by some implementation details. The following chapter (Chapter 5) presents the the evaluation we have made on our system, presenting first some examples of its functionality and the results from tests of performance. The last Chapter (Chapter 6) is the conclusion, were we wrap up our work and describe the future work.

Chapter 2

Concept Description

The following sections will provide an overview of several topics that are related to this work. It starts by explaining the current technology, followed by the presentation of a couple of processing techniques that are commonly applied to improve performance. Finally, present some generative design techniques that our system should support.

2.1 Generative Design

Procedural generation is the algorithmic generation of content instead of the usual manual creation of content. This can be applied in almost all forms of content, but is mostly used in the generation of graphic content, such as textures, geometry and animations, in which is included generative design. Procedural generation is also used for the generation of sound, with procedurally generated music and synthetic speech. **Generative design** is the application of procedural generation to design, i.e., any design practice where the designer uses a computer program to produce a solution to a design problem[9][10].

The key property of generative design is that it describes the data entities, such as textures, geometry, or sounds, in terms of a sequence of instructions rather than as a static block of data [11]. This allows the production of big volumes of detailed, high quality, graphic content without the costs, both in time and money, of manual content creation. Since it is based on procedures, it provides parametric control. Users can introduce in their programs as many useful parameters as they want, which allows them, for instance, to have different results from just one implementation by just changing some control variables.

2.2 Graphic APIs

There are several Application Programming Interfaces (APIs) for graphic content creation, but the most well-known ones are DirectX and OpenGL.

DirectX is a collection of multimedia APIs created by Microsoft for their platforms. It includes the Direct3D API. This tool has evolved very much since it was released and supports state-of-the-art techniques such as hardware acceleration and so forth. On the other hand this system is only supported by Microsoft platforms and since this work should not be limited to the Microsoft platforms, these tools will not be used.

OpenGL is an open-source library that is widely used. This system will be better explained in the next section.

There are other APIs for graphics such as Vulkan [30] that is being described as the OpenGL replace-

ment and Metal[1] which is the Apple's API and was developed to run on their platforms, MAC OS and iOS.

2.3 Modern OpenGL

OpenGL is a well known cross-platform API created by Silicon Graphics Computer Systems with Version 1.0 released in July of 1994 for 3-D Graphics and Imaging. It is a streamlined and hardware-independent interface that can be implemented on many different types of graphics hardware. It is also independent of the machine's operative and windowing systems.

Major changes has been imposed to this library from its early versions and this section covers the modern version of OpenGL versions $4.x$.

OpenGL provides a small set of geometric primitives - points, lines, triangles and patches that are specified by their vertices. From this set of geometric primitives all geometry is constructed, both in 2D and 3D.

There are some steps that are performed to render an image, and OpenGL follows the pipeline in Figure 2.1. While some of the steps are fixed and are automatically executed, other steps are programmable, which allows the developers to directly program to the GPU. This code that runs on the GPU is called shader. Shaders can be thought of as small programs that are specifically compiled for the GPUs[3].

We now describe the steps. First the model is created from geometric primitives and it is the input for the pipeline (*Vertex Data* on Figure 2.1).

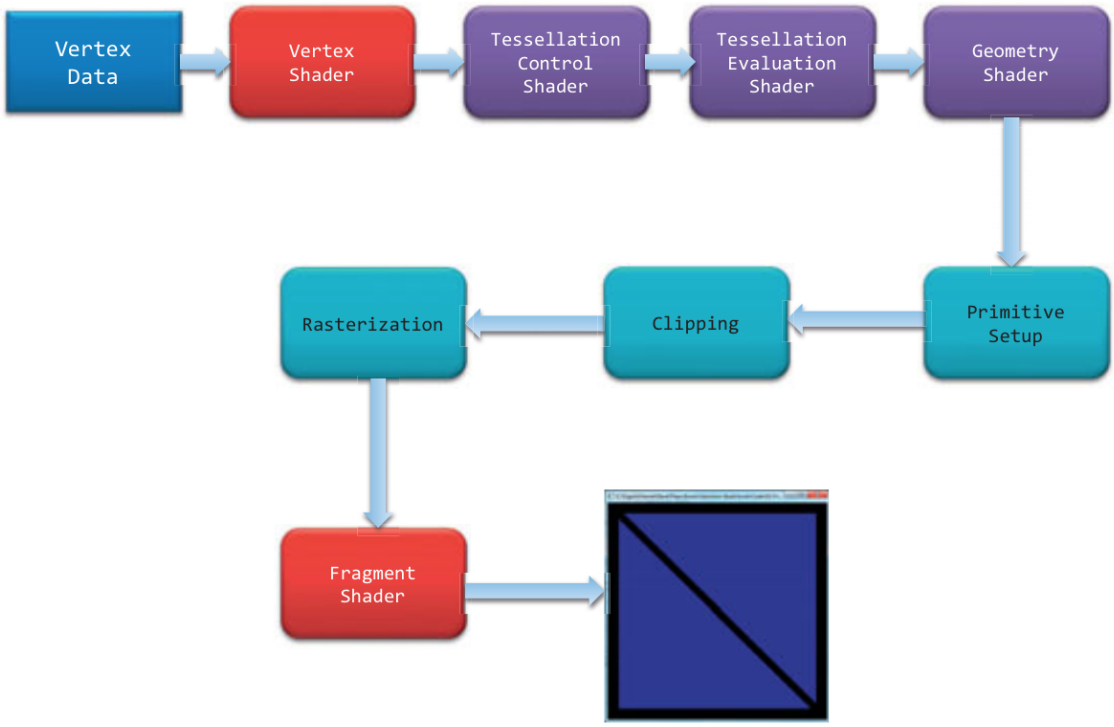

Figure 2.1: OpenGL Pipeline [3]

The first step of the pipeline is the Vertex Shader that processes the data associated with each vertex.

After, there are three optional shaders. In these three there are two Tessellation Shaders. With these shaders simple geometries can be tessellated and increase the number of primitives to improve the models dynamically.

[1] `https://developer.apple.com/metal`

The third optional shader is the Geometric Shader that allows the additional processing of geometric primitives and also including the creation of new primitives.

Until now all steps work with vertices. After those steps there are three fixed steps, primitive assembly, clipping and rasterization that assembly the vertices into primitives, clip the geometry cutting the parts that falls off the "screen" and the generation of fragments, respectively.

A fragments is a *candidate pixel*, in that pixels have a home in the framebuffer, while a fragment can still be rejected and never update its associated pixel location [3].

2.3.1 Vertex Shaders

Vertex Shaders can be very simple, from a *pass-through shader* that just copies the data to the next step to very complex ones.

These shaders are used to perform computations that calculate the position of the vertices in screen coordinates, assign vertex's color using lightning computations, etc..

Vertex Shaders have some limitations, they cannot create additional geometry and cannot access data of other vertices. They can just process the data of the current vertex so the number of vertices after this step is the same as before.

2.3.2 Tessellation Shaders

Tessellation Shaders are very different form the previous ones. This shaders address some of limitations presented before. This shaders work with a geometric primitive called a *patch*. A patch is a list of vertices that preserves their order during processing. Each patch can have an arbitrary number of vertices that have to be specified before drawing, in contrast to the other primitives that have a specific number of vertices.

2.3.3 Tessellation Control Shader

Tessellation Control Shader defines the layout of the output through the generation of the tessellation output-patch vertices and the specification of the tessellation level factors. The output-patch vertices is the list of vertices that results after the input vertices have been processed. The tessellation level factor defines how much the output patch is tessellated.

OpenGL supports three tessellation domains: a quadrilateral, a triangle, and a collection of isolines [3]. To control the amount of tessellation two sets of values are assigned, the outer-tessellation values and the inner-tessellation values. This values define how the perimeter or the interior of the domain are subdivided respectively.

As an example, Figure 2.2 shows a triangular domain with the following tessellation levels.

```
gl_TessLevelOuter[0] = 6;
gl_TessLevelOuter[1] = 5;
gl_TessLevelOuter[2] = 8;

gl_TessLevelInner[0] = 5;
```

In this example, three outer control values are used, each one corresponds with one side of the triangle and one inner value. Each outer value defines the number of divisions that its correspondent side has.

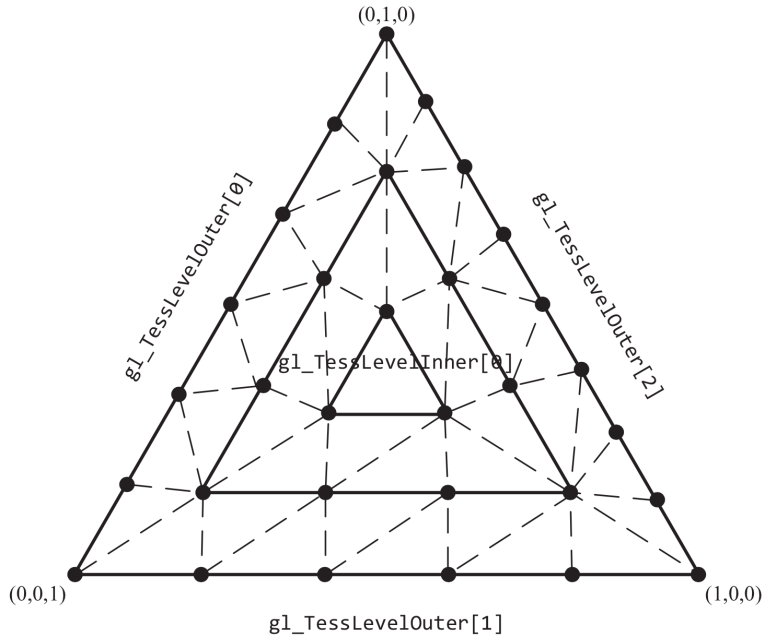

Figure 2.2: Example for tesselation control with triangular domain [3]

2.3.4 Tessellation Evaluation Shaders

Tessellation shaders work with the output of the previous phase. Here the vertex positions are computed from the tessellation computed before. It is is basically responsible for the computation of the vertices' screen positions from the layout defined.

2.3.5 Geometry Shaders

Geometry Shaders are the first shaders that access the complete primitive as a list of vertices and with that it is allowed to do different actions that require this access to information. The amount of output can be variable so both *culling geometry* and *geometry amplification*, respectively, output less vertices that the input and, output more vertices than the input. Also, in this shaders, the primitives type can be modified, i.e. the input can be *quads* and the output be a *triangle_strip*.

Geometry shaders however have a limitation. Each call of a geometry shader have a maximum number of vertices that it can output. This limitation could be important for the implementation of geometry amplification. This maximum number is hardware dependent and varies with the size of the output buffer that is used by the GPU to support geometry shaders. OpenGL specification since version 4.3 imposes 256 as the minimum number of vertices supported.

2.3.6 Fragment Shaders

This shaders implement the last phase of the pipeline. Here the fragment's final color is computed and also the depth value.

Fragment Shaders are useful to implement texture mapping or lights, for instance.

2.4 Processing Techniques

Within the following section, a number of techniques that can be applied to improve performance of the processing and visualization of large amounts of geometry will be presented. First the Level of Detail

(LOD), Section 2.4.1, which manages the detail that each object is generated with and, after, Culling (Culling) that manages which objects are generated.

2.4.1 Level Of Detail

Level of Detail is a technique that is used to improve the performance of the graphic pipeline. This is done by managing the complexity of the objects representation relative to some indicator. Within this indicators, the most common one is the distance of each object to the viewer. If an object is far from the viewer a decrease on the detail will not be noticed and will save computation time. Other indicators can be the importance that is assigned for each object, relative speed or partial occlusion.

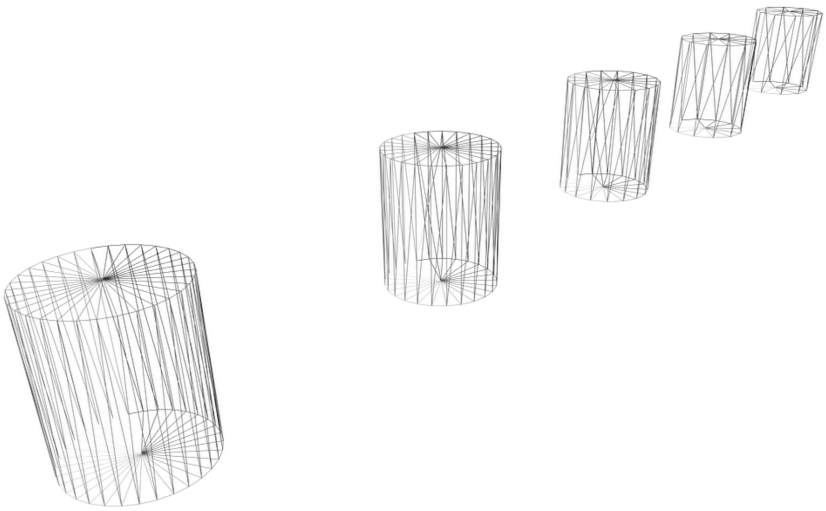

Figure 2.3: LOD example

This concept is easy to understand and implement if we look at the example in the Figure 2.3. In this figure there are five cylinders that have different detail according to the distance to the camera. In this case only the number of sides of the cylinder changes.

2.4.2 Culling

Culling is a set of techniques that are used to improve performance. It involves determining which parts of the model do not have any contribution to the final image, and with that information reject this objects from the pipeline. Examples of this techniques are *backface culling*, *view frustum culling*, and *occlusion culling* [12]. Backface culling culls the polygons that are not facing the camera, view frustum culling eliminates the objects that are out of the view frustum, and occlusion culling eliminates the objects that are occluded by other objects.

These techniques are implemented in different stages of the rendering pipeline. It can be implemented by the user within the models description, so that it only runs the code that generates visible objects. However it is hard to implement and, therefore, is neither a good or popular solution specially for our target users.

Most of the 3D Engines implement Frustum Culling at the software level testing the objects before they are sent to the GPU [13].

Oclusion Culling is the most complex technique. It requires to generate the entire model and compute how each one's visibility is affected by the rest of the model. This can be done automatically by the GPU using a technique known as *Z-Test*. This technique uses a buffer, *Z-Buffer*, to store the z-values for

each pixel and with that information compute which object is fragment is visible. However this solution is not great because it is applied only after all the model is generated. Other techniques sort the objects by their distance to the camera and with that information compute the non-visible objects.

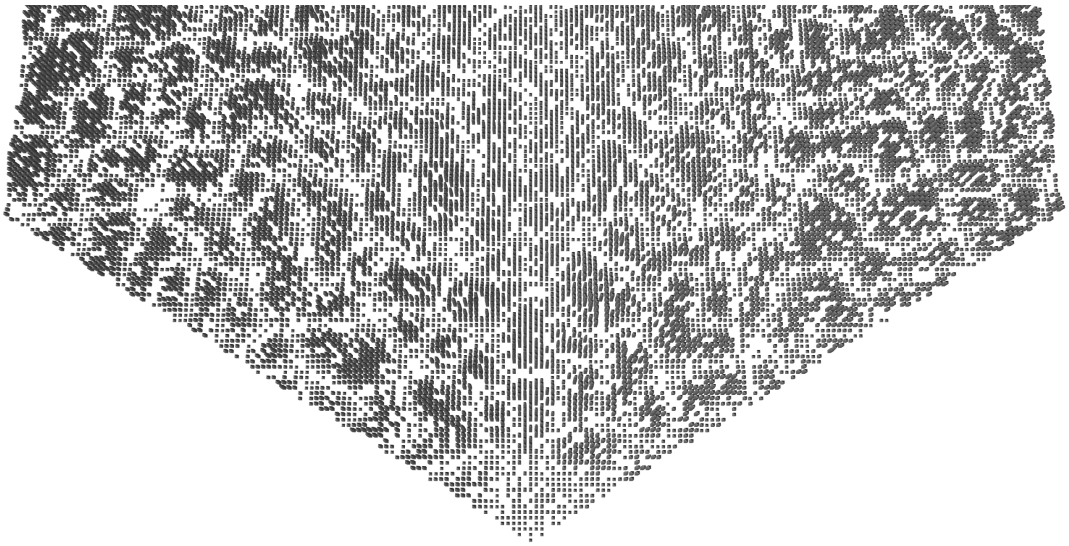

Figure 2.4: Object Culling example

Most of these techniques are applied after processing the objects, but if this concept is applied before the generation of the objects, thus preventing the input of large amounts of geometry through the pipeline, we can make a large improvement on performance. Figure 2.4 is a good example. It represents a city but where only the buildings that are visible from the current position of the camera are generated. This test is dynamically done, before the generation of each object and is not hard-coded within the model description.

2.5 Generative Design Techniques

In the following sections some generative design techniques are explained. These techniques are applied to the procedural generation of various types of forms.

2.5.1 Fractals

A fractal is defined as "a geometrically complex object, the complexity of which arises through the repetition of a given form over a range of scales"[14]. This concept is observed in some forms that exist in nature. Trees, mountains, coastlines and the network of neurons on a human cortex can be seen as examples of fractals. Natural shapes tend to be irregular and fragmented and exhibit a complexity incomparable to regular geometry [15]. Fractals were proposed to be seen as a new form of symmetry [14], *Dilation Symmetry*, which is when an object is invariant over a change of scale. This invariance might be only qualitative and not exact. For instance, a river network exhibit dilation symmetry if *zooming in* in some part looks the same as the whole image. As this example, many others show dilated symmetry such as clouds, tree branches and some vegetables as shown in Figure 2.5.

This idea was applied in maths and resulted in a new area called fractal mathematics. The objective of this field is to describe very complex shapes with simple rules such as repeating a substitution pattern.

Figure 2.5: Fractals in Nature

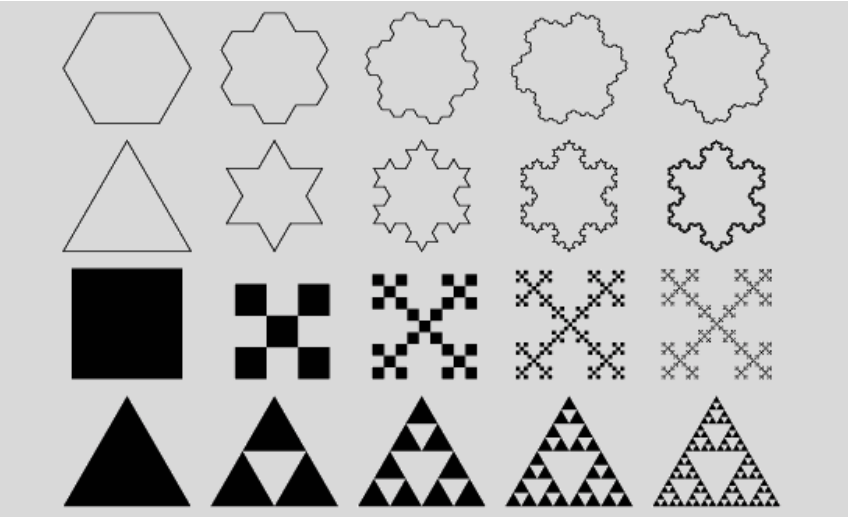

Figure 2.6: Geometric Fractals

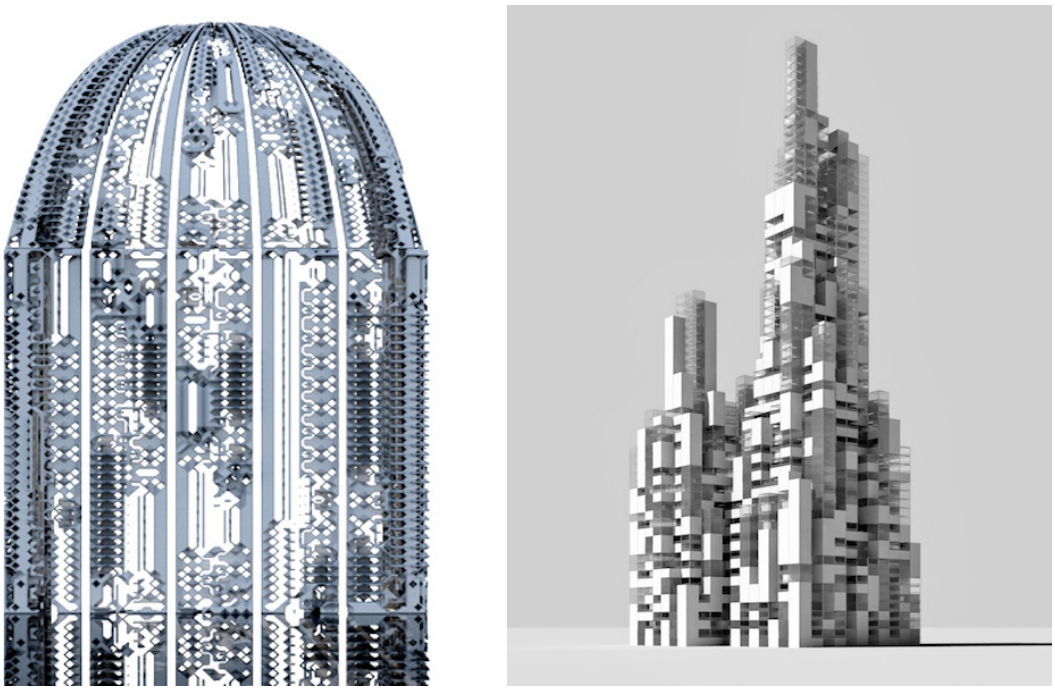

Figure 2.7: Examples of cellular automata applied in architecture

In Figure 2.6 there are four examples of Geometric Fractals, with the first five iterations of each one. All of them are built by the substitution of a part of the image by another one.

The example of the second row is known as the Koch snowflake. In this example, in each iteration, all the line segments are replaced by four segments with 1/3 of the size of the original one with the two in the middle being placed in a angle forming a equilateral triangle with the original line that is removed.

It is clear that the detail that is presented in each iteration increases as the scale changes. There is the concept of *Fractal Dimension* that tries to measure this evolution, in which the detail in a pattern changes in comparison with the scale in which it is measured.

As stated before, the world is visually very complex, so when synthesizing worlds, "**complexity** equals **work**"[14]. This work can be done by the programmer/artist or by a computer. Fractals as being defined as a simple mathematical function, it is relatively easy to implement a procedure that model one fractal.

This technique is often used to model many natural forms that present fractal properties. Mountains, for instance, are usually modeled using of fractals. Other natural forms that present fractal properties are trees, river systems, lightning or vascular systems in living beings. Fractals can also be used in architecture to implement organically looking textures or shapes.

2.5.2 Cellular Automaton

A cellular Automaton is a model of a system of cells within a grid with a given shape, each of this cells can be on one of a finite set of states. It evolves during a finite amount of time-steps with a set of simple rules according to the state of the neighboring cells. The neighborhood of the cell can be defined in many different ways, the most common is the use of the adjacent cells.

This models have various applications, such as modeling of nature aspects (Figure 2.10), textures, and as inspiration to architecture (Figure 2.7).

The case where each cell have two possible states and the next generation state depends only on the

previous state of the cell and the two immediate neighbors is called an *elementary cellular automaton*. In this case we have $2^3 = 8$ possible patterns for a neighborhood and $2^8 = 256$ sets of possible different rules. These rules are referred by their *Wolfram code* [16].

A common initial state for this elementary cellular automata is a random line. To be able to compare the results between rules and get clean results another option is to start with a line with zeros except the middle cell that is initialized with the value one. Applying this second option and the set of rules in Figure 2.8 (the rule 30), we get the pattern in the Figure 2.9 that represents the evolution of a cellular automaton over a few generations.

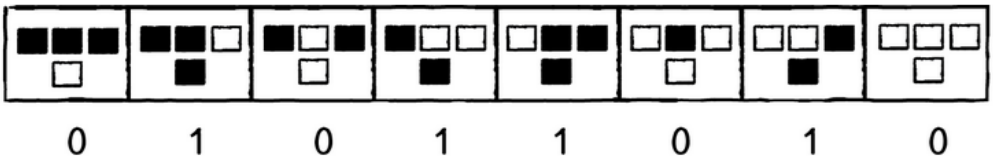

Figure 2.8: Example Production Rules [4]

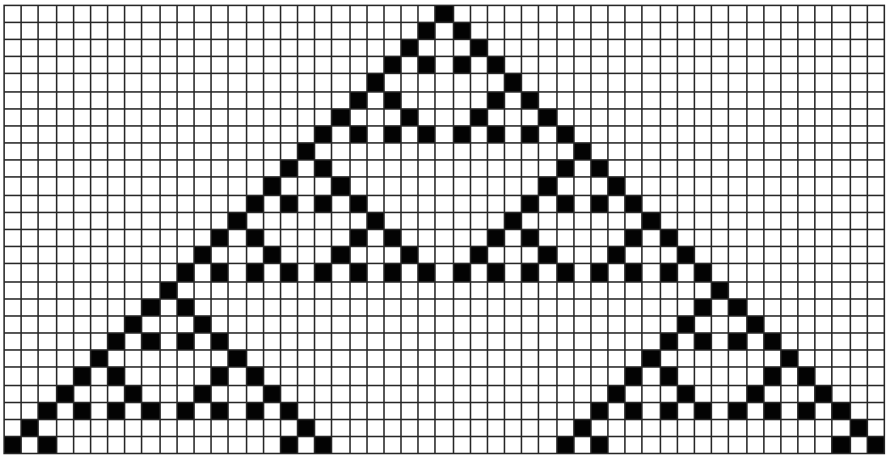

Figure 2.9: Sierpiński Triangle, rule 90

In Figure 2.9 each line represents an iteration of the system with the application of the rules. With this set of rules a Sierpiński triangle is reproduced.

Cellular automata are used mainly to model phenomena that occur in the physical world, most of them can only express the basic idea of a phenomenon, but some are accurate enough to be able to make predictions [14].

In this context, cellular automata are used to model natural shapes and textures, Figure 2.10 shows on the left, a natural texture on the shell of a *Textile Cone Snail*, that looks like the patterns formed with the cellular automaton on the right.

2.5.3 L-Systems

Lindenmayer Systems (L-Systems) are a class of string rewriting mechanisms, originally developed by Lindenmayer as a mathematical theory for plant development. It is capable of describe the behavior of plant cells and model the growth processes of plant development.

An L-System consists of two different parts, one axiom and a set of production rules. The axiom is the starting point of the system, acting as a seed. Then it is applied in this seed the set of production rules, that change the initial string, producing other strings. This is an iterative process, so after the

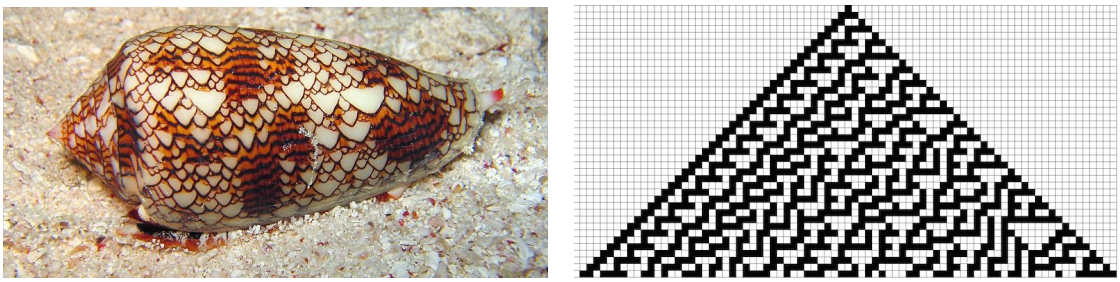

Figure 2.10: Example of the representation of natural patterns with cellular automata. On the left, a Natural Shell [4] and on the right a Pattern formed with the rule 30.

production of a larger set of strings, the rules can be applied to each one of them which grows the size of the set even more.

L-Systems are used to model the natural growth of vegetation (Figure 2.11), and the generation of Fractals.

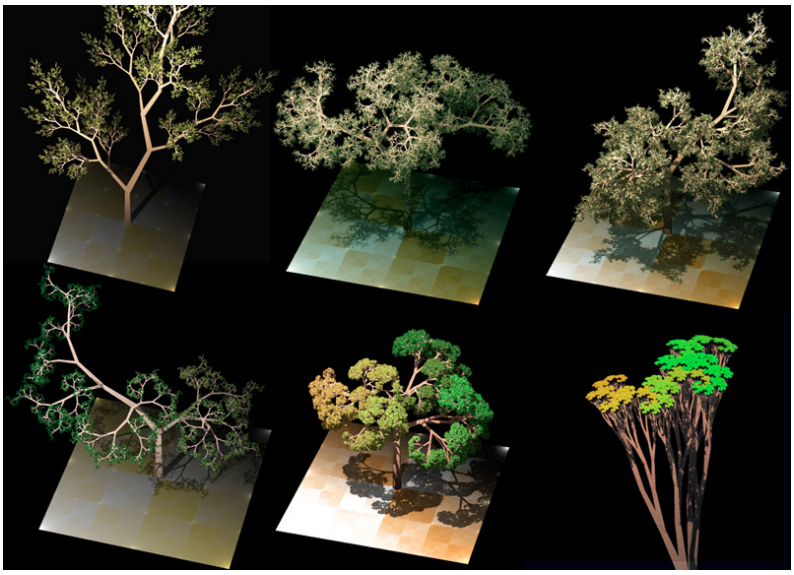

Figure 2.11: Trees with L-Systems

In this process, each symbol is associated with a production rule. For instance having $\{F, +, -\}$ for the alphabet and *production* $\{F \rightarrow F + F - -F + F\}$, from a starting axiom *aba*, and the application of the rules we have:

$$F \tag{2.1}$$

$$F + F - -F + F \tag{2.2}$$

$$F + F - -F + F \ + \ F + F - -F + F \ - \ - F + F - -F + F \ + \ F + F - -F + F \tag{2.3}$$

This is an example of the evolution of one system where the production is applied in (2.1) that turns into $F + F - -F + F$. Note that the space between the symbols are just for readability.

All the symbols are assigned with a geometric meaning. The notion of a turtle with a pen, as proposed in [17], with the symbols being interpreted as moving instructions to the turtle, is a simple way to understand, where "F" means forward and the symbols "+" and "-" are interpreted as rotations counter-

clockwise and clockwise respectively by a predefined angle. By applying this method to the last example and setting the angle for the rotation to $60°$ the result is Figure 2.12.

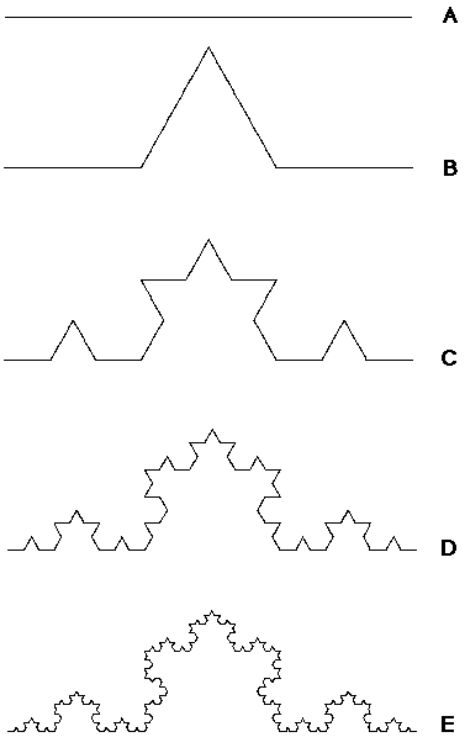

Figure 2.12: Result of the "turtle walk" with the given example

2.5.4 Shape Grammars

Shape Grammars[18] can be considered grammars for design. Instead of having symbols or letters as components of the alphabet, it has shapes that can be in 2D or 3D, and has production rules that are composed by these shapes, that specify the evolution of the system. With this process, similar to the L-Systems explained before, the shape starts from a seed, i.e. a usually simple shape and can evolve to one big and/or complex shape.

The process is performed in two steps, the recognition of a shape and the replacement according to the rules previously defined.

Figure 2.13 exemplifies a simple shape grammar with one rule, and the Figure 2.14 presents the derivation of this grammar iteratively. In this image, it is shown that from very simple initial shape, a complex from can be generated after a few iterations.

In the CityEngine [2] system (Section 3.3), this technique is applied to the generation of buildings using 3D blocks for the main form, and 2D shapes to design the facades. Figure 2.15 shows a simple building that I modelled using CityEngine and its CGA Shape Grammar. However, CGA is powerful enough to model much more complex buildings like the one in Figure 2.16.

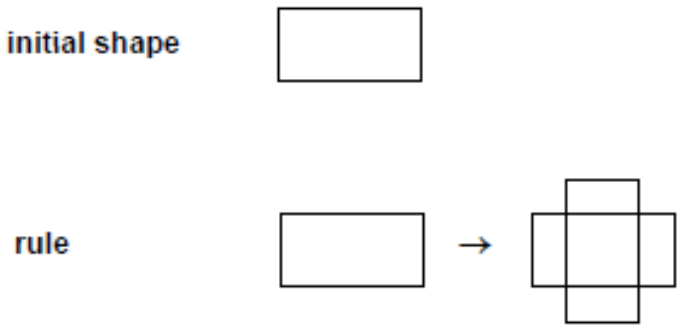

Figure 2.13: Shape Grammar initial shape and rule

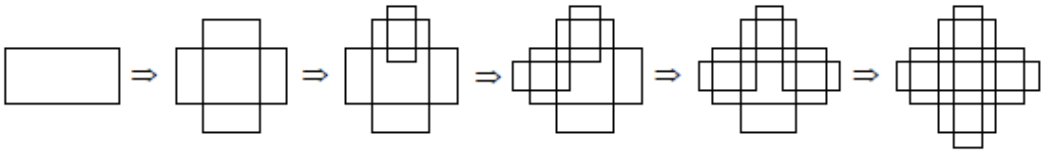

Figure 2.14: Derivation of the rule in Figure 2.13 with 5 iterations

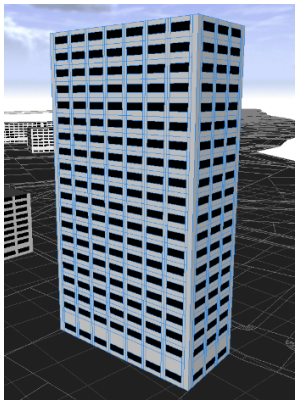

Figure 2.15: Simple Building

Figure 2.16: Complex Building [2]

2.5.5 Noise

To generate irregular procedural textures, we need an irregular primitive function, usually called noise [14]. It is a pseudorandom function that breaks the monotony of a pattern and it look more random. Perlin Noise is the most well-known and used noise function. It was created by Ken Perlin, for the movie Tron to generate natural looking textures.

The psedorandom property is important and a true random function like *white noise* would not do the job. If we generate a texture based on white noise the pattern would change every time it is generated and we would like that it stays the same, frame after frame. This is achieved with the use of inputs for this function so that the same input always returns the same output sequence.

With this noise function, we can generate a sequence of values that are interpolated to produce a coherent noise. With the application of *turbulence*, that is composition of several layers of this noise with different frequencies and amplitudes. These layers are called *Octaves* and the ratio between amplitude and frequency of the layers can be expressed as a constant known as *persistence* [19]. With the result we can create a texture that looks natural and with fractal like structure.

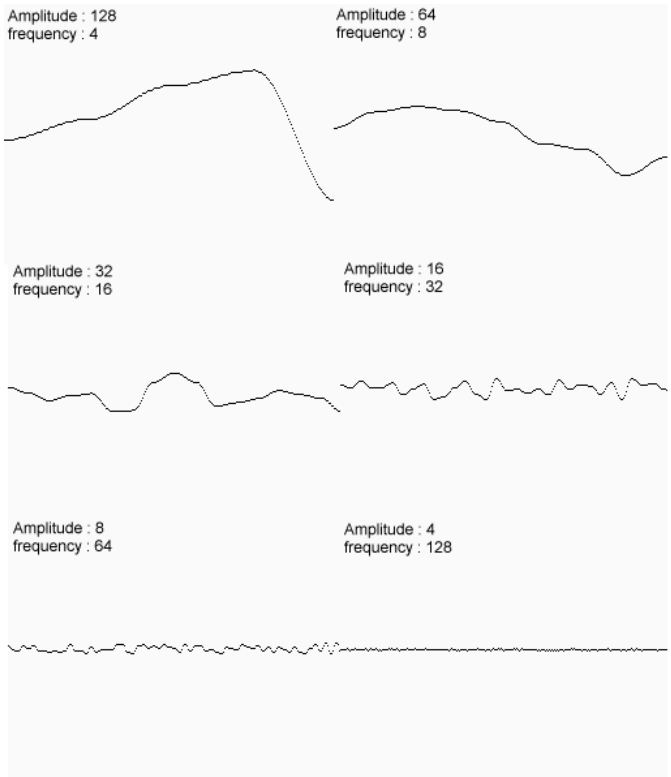

Figure 2.17: Different noise functions

For instance, the Figure 2.17 shows the result of the interpolation over six noise functions with different frequencies and different amplitudes. And the sum of all this functions is illustrated in the Figure 2.18 [20].

Noise can also be used to generate planes. The method used is the same as the 1D problem but we have to generate a lot more data points that are then interpolated as a plane. This results in noisy images that are often used to model clouds, smoke and other textures with similar visual properties as illustrated in Figure 2.19. Another application for this technique is the generation of height maps.

Another application for Noise planes is *object placement* on a grid. By creating a noise plane with the same size of the grid, with each cell of the grid corresponding to one pixel of noise, the object placement is done by choosing each object for each cell according to the noise value. Figure 2.20 shows a city were

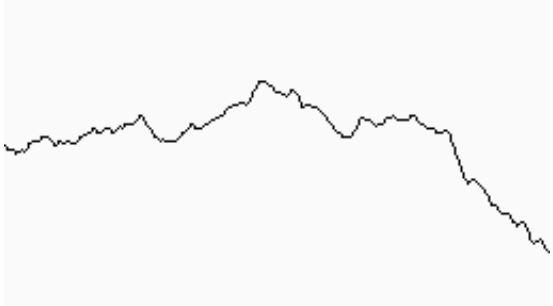

Figure 2.18: Combined noise functions
"You may even imagine that it looks a little like a mountain range."

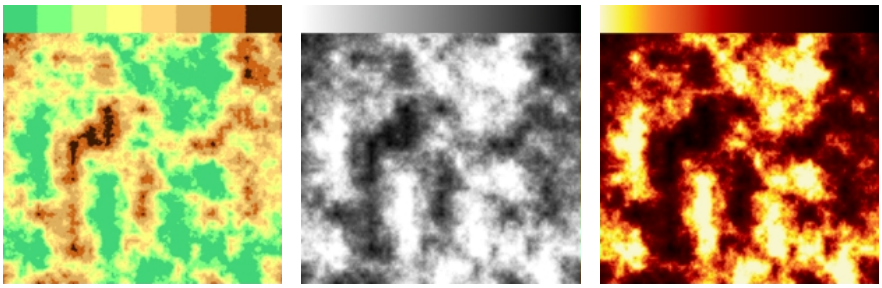

Figure 2.19: Gradient mapped textures [5]

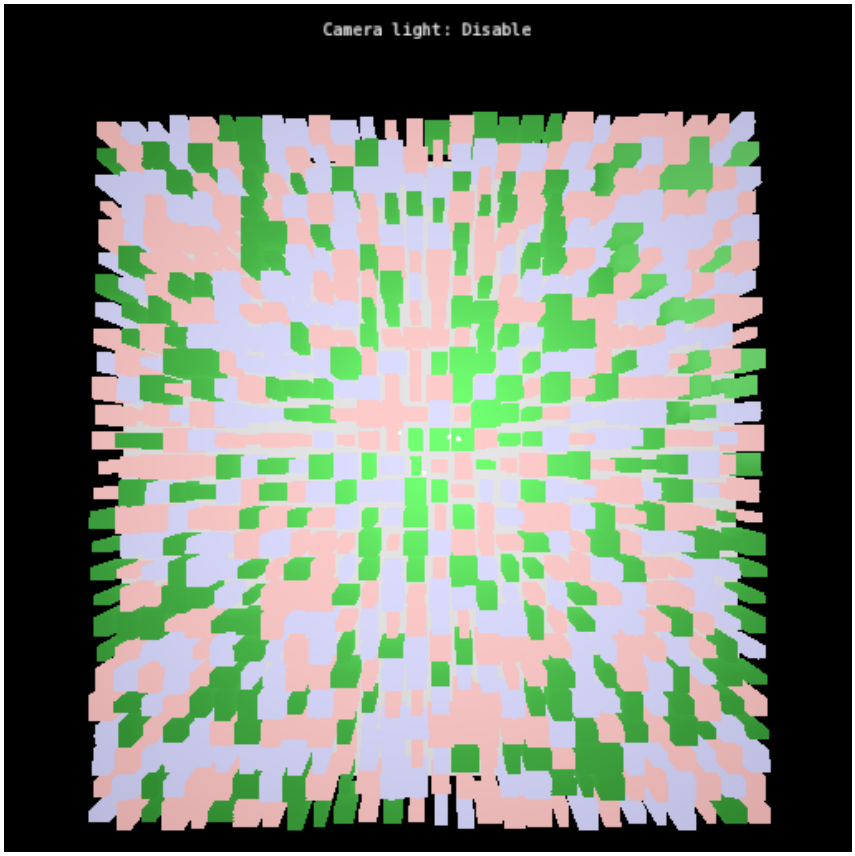

Figure 2.20: Objects Placed following a noise function

the buildings where placed with the use of a noise plane. In this cases the noise domain was splitted in three intervals, each one corresponds to one type of building (commercial, industrial or residential). After setting the type for one block, the system randomly chooses one from a set of buildings of that type.

Chapter 3

Related Work

This section presents a number of systems that have goals that are similar with ours. They are good sources of ideas and points of comparison to evaluate our solution. The described systems are the most relevant ones, that take in consideration the field of architecture and generative design.

3.1 VR Azuchi Castle

In [21], Tomohiro Fukuda et al, presented a High-Definition (HD) Virtual Reality (VR) system for Historical Architectural and Urban Digital Reconstruction. They used the Azuchi Castle and the surrounding town as case study.

They have the goal of having a real time rendering system with high accuracy and realism. With that, they face the same problems as we did. In their work, they consider the problem of rendering large sets of objects of different scales in real time. It is also presented the various techniques they used to face this problem.

Figure 3.1: Azuchi castle view

The most largely used technique is Level of Detail management. It is described the application of this technique on the roof tiles and the terrain texture mapping.

This model have two types of tiles, flat and round, where the second have 2.2 times more polygons than the first. The management of the level of detail consists in the definition of three distance intervals between camera and the model. In the first, between 0 m and 50 m, both tile types are rendered. From 50 m to 100 m the proportion of the round tile is linearly faded. After 100 m, only flat tiles are rendered.

A similar technique is used to the terrain texture mapping problem, which is related to the amount of texture data is needed and is dealt as by having different textures that are chosen according to the distance to the camera.

3.2 Undiscovered City

In [22] Stefan Greuter et al. presented a system that generates in real-time pseudo infinite virtual cities which can be interactively explored from a first person perspective. In their approach "all geometrical components of the city are generated as they are encountered by the user." As shown in the Figure 3.2 only the part of city that is inside the viewing range is generated. This method allows the visualization of massive amounts of geometry, buildings in this case, by generating in real time only the geometry that is on sight, and since this subset is usually much smaller than all the geometry this results in high performance benefits.

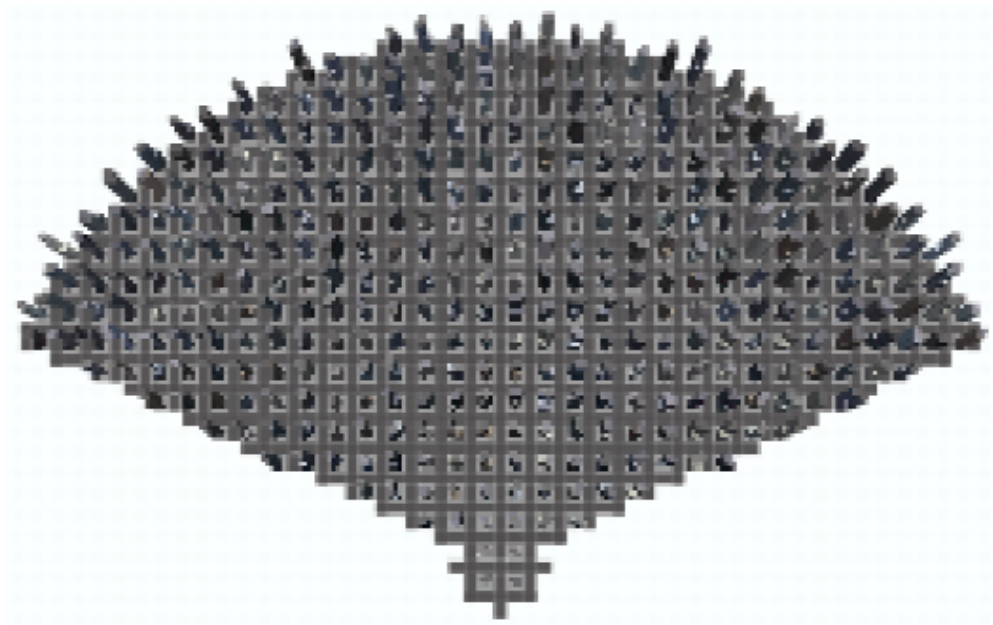

Figure 3.2: Viewing Range

The system uses a 2D grid that divides the terrain into square cells. The cells represent proxies for the content that will be generated procedurally. Before the content of each cell is generated, the potential visibility of it is tested. After that, only the visible cells are filled with content.

Then the roads are created in a uniform grid pattern. This grid does not feel natural. Thus, to make this system more realistic, it joins some of the grids to create a less uniform distribution of the buildings.

The buildings within this project are generated with the simple extrusion of regular polygons. These extruded forms are composed to create complex architectural models.

3.3 CityEngine

CityEngine [1][2] is a three-dimensional (3D) modeling software developed by Procedural Inc. (now part of the Esri R&D Center), specialized in the generation of 3D urban environments. With the procedural modeling approach, CityEngine enables the efficient creation of detailed and large-scale 3D city models with a lot of control from the user. This system applies the concept of Immediate Feedback by allowing the user to immediately see the results of each change. This is implemented through a set of sliders (Figure 3.3 top left), that are assigned to various indicators in the model. This indicators can be as high level as the *size of the city* or as specific as the *width* of a window or the *number of floors* in a building.

The following sections explain how CityEngine faces each of the steps in the generation of a city, such as road network generation or building modeling.

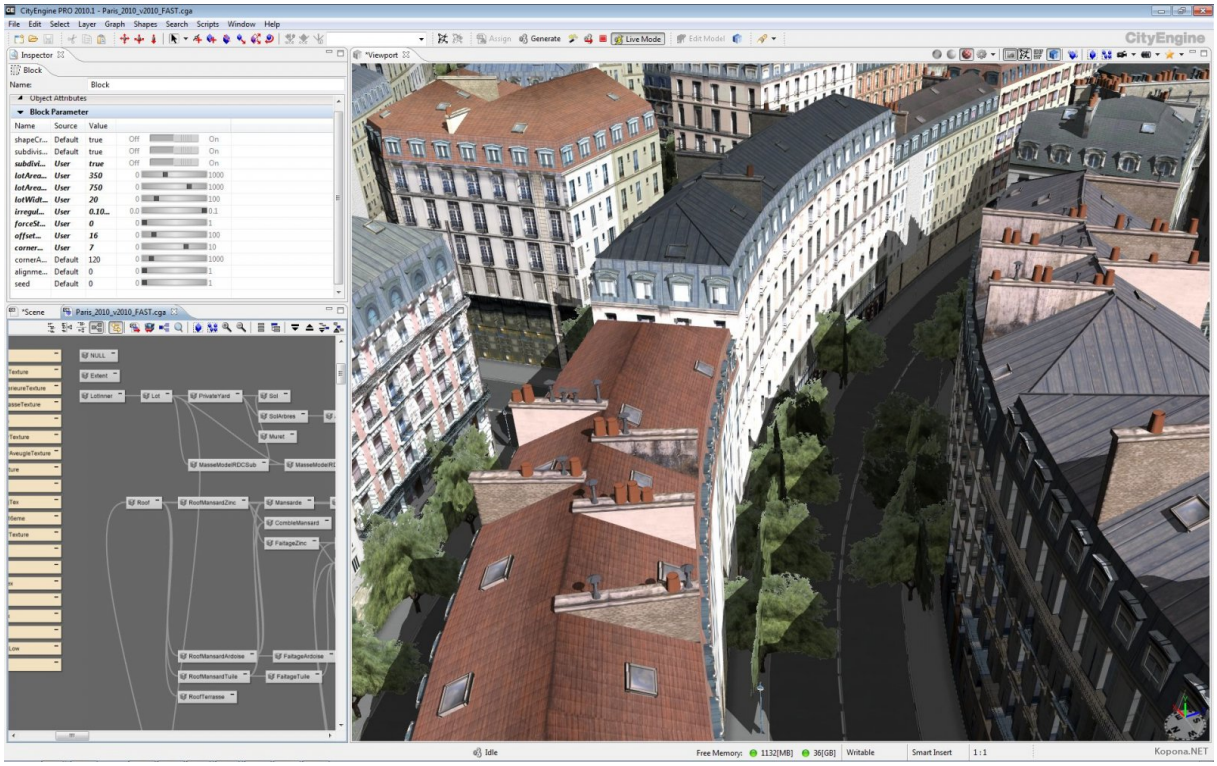

Figure 3.3: City Engine Interface

3.3.1 RoadNetwork

The first part to procedurally generate a city is to create a road network to become a backbone of the city and provide an overall structure. For that, CityEngine receives as input maps such as land-water boundaries and population density. From that input a network of highways is created to connect the areas off high density population, and small roads connect to the highways. This growth process continues until the average area of each lot is the desired one. The system have a default value for the average lot's area, but it can be set by the user to a different one.

To implement this growth process, it uses an L-System (Section 2.5.3) that computes the road network.

The Figure 3.4 shows the evolution of this process in a map of Manhattan. The first two pictures on the top shows the process in different phases during the process, the picture in the middle is the result of the process and the bottom line is the real map of Manhattan for comparison.

Figure 3.4: Road Map growth

3.3.2 Buildings

To implement the generation of buildings, CGA was created, which is a shape grammar (Section 2.5.4) that was introduced in [1]. It is defined as "a novel shape grammar for the procedural modelling of CG architecture, produces building shells with high visual quality and geometric detail." To do so, this grammar uses a group of well defined production rules.

This tool allows the user to model buildings with high control and in different ways. It can be done by text, writing production rules from a shape grammar or with a visual language similar Grasshopper 3D, that is nice for simple models but it is hard to work with more complex models, for instance, Figure 3.3 shows a set of rules (bottom left), that is relatively small but is already difficult to follow the connections between rules.

Mass Modeling To model a building the first step is to create a mass model of the entire building by assembling basic shapes. With scaling, translation, rotation, and split applied to basic shapes namely I, L, H, U and T as shown in the Figure 3.5.

The next step is to add the roof, from a set of basic roof shapes or general L-Systems.

After that, with the application of the grammar rules in the created mass, it is possible to create complexity to the level that is desired, being able to produce highly complex buildings like the one in Figure 3.6.

3.3.3 Results

The result can be a city like Figure 3.7, with approximately 26000 buildings.

Figure 3.5: Basic shapes

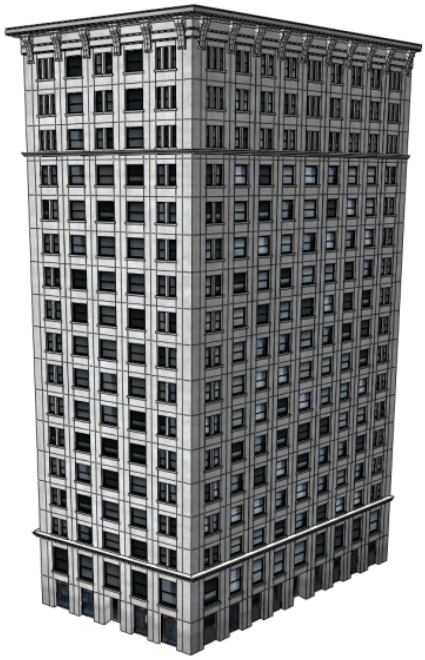

Figure 3.6: Complex building modeled with CGA

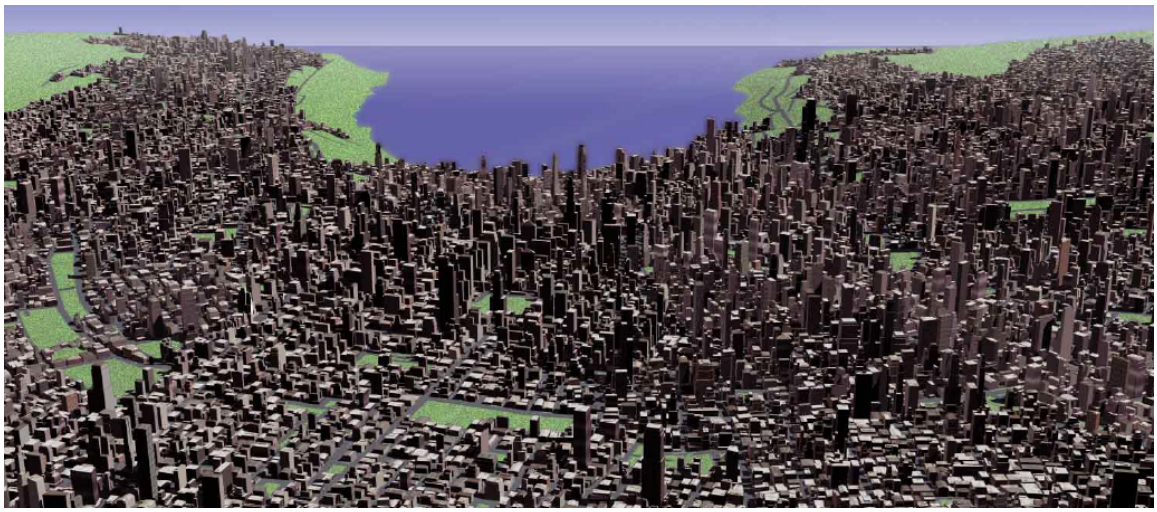

Figure 3.7: City with approximately 26000 buildings.

CityEngine outputs can be imported by Maya[1], to achieve better results. Figure 3.8, that represents a 'virtual' Manhattan generated by CityEngine.

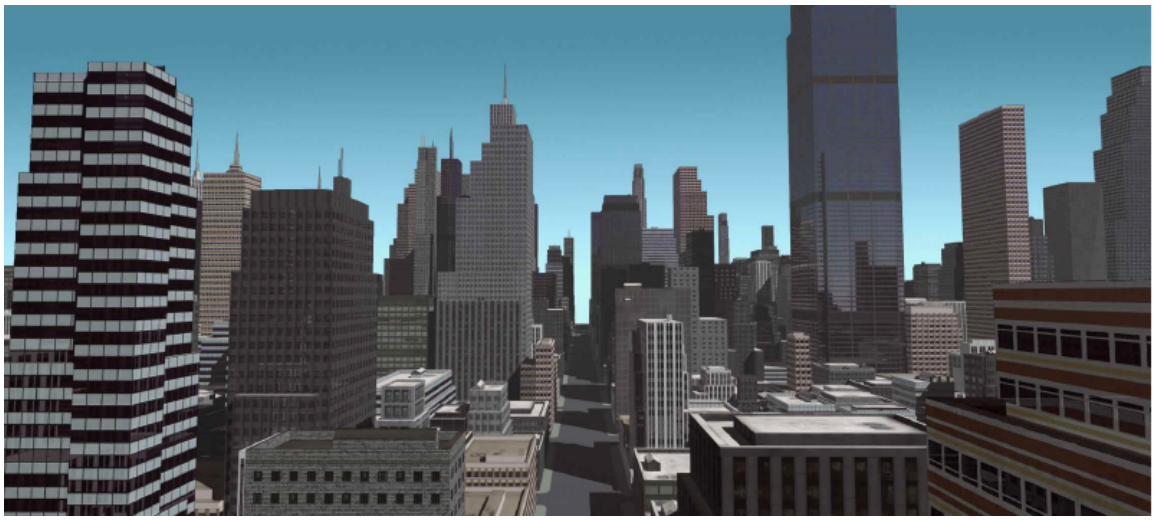

Figure 3.8: City rendered with Maya.

3.4 No Man's Sky

No Man's Sky[2] is a game being developed by indie studio Hello Games. It is an exploration game in an universe entirely procedurally generated and has 1.8×10^{19} planets. Some of the planets are populated with a set of flora, fauna and its behavior, all that also procedurally generated. This is achieved by providing a seed number to a deterministic engine that generates a pseudorandom sequence of numbers that is used to generate always the same universe. This is important due to the fact that every player will find the same planet, with all the same features and population given only the planet coordinates.

Despite the enormous size of the universe, only the planet that is currently being explored is generated, thus achieving good performance. Figures 3.9 and 3.10 are screenshots of the game.

Figure 3.9: Screenshot of *No Man's Sky*

[1] `http://www.autodesk.com/products/maya/overview`
[2] `http://www.no-mans-sky.com/`

Figure 3.10: Screenshot of *No Man's Sky*

3.5 Computer Aided Tools

CADs are very powerful tools for design and architecture. With these tools, architects and designers are able to create, interact, and visualize models. CADs are developed to be interactively used. Although the users create the models in a manner very close to drawing on paper, these tools provide a growing set of commands for them to automatize their work. They are very good to analyze and visualize models, Figure 3.11.

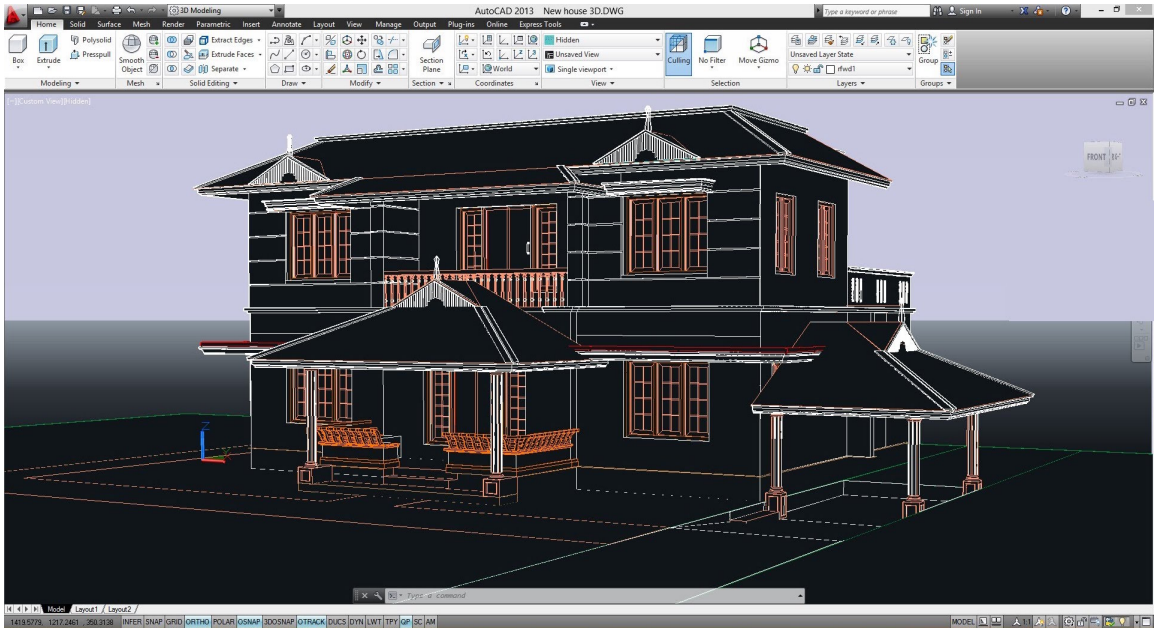

Figure 3.11: Autodesk Autocad Screenshot

Most of the CAD tools also support GD providing APIs for several PLs, such as AutoLisp, C#, and more recently Ruby. But, since these tools were designed for manual use, they are not prepared to deal with the speed that GD is usually able to create large amounts of geometry. This causes significant losses in performance for these tools.

Another problem is that the functionality that each tool provides is different, and also the APIs are different even for the same functionality. This raises portability issues since GD programs written for

one CAD tool will not work with other one.

3.6 Grasshopper

Grasshopper[3] is a Generative Design tool that explores the visual programming paradigm. It was developed as a Rhino[4] plug-in, and it have been highly accepted within the designer and architecture communities, since it makes it very easy to start modeling without much prior coding knowledge and to create simple programs. It happens that, as most visual languages, when the user starts to create complex models, it gets very confusing and difficult to understand the programs. Figure 3.12 is an example of a Grasshopper program that is difficult to understand.

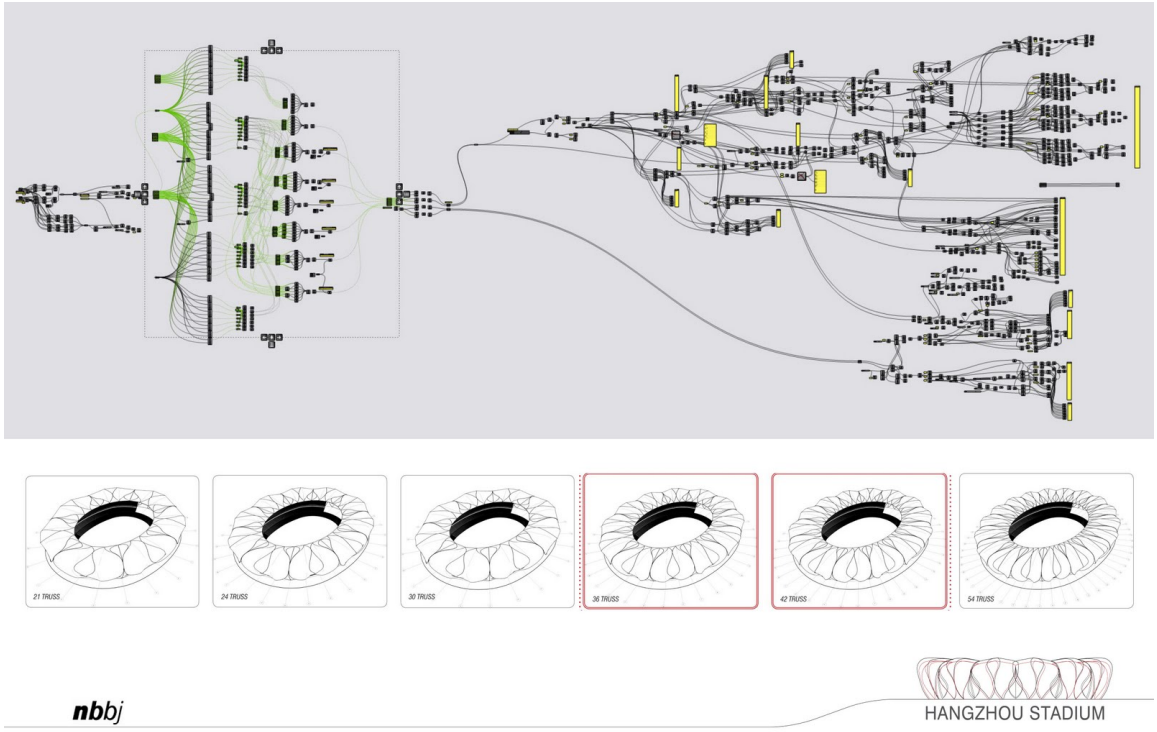

Figure 3.12: Complex Grasshopper Model

3.7 Building Information Modeling

Building Information Modeling (BIM) is a recent development in the fields of architecture, engineering and construction. This technology creates an highly accurate virtual model of a building [23]. This model, known as building information model, contains much more than geometric information, that is provided by as CAD tools. It contains objects with semantics that includes information such as geometry, spatial relationships, parametric rules and properties such as materials, finishes, manufacturer and costs. For example, it allows the users to choose a specific door from one provider with all its data, rather than manually drawing a door with all its components [24].

This technology brings functionalities such as visualization, overall cost estimation, collision detection or even building management.

[3] `http://www.grasshopper3d.com/`
[4] `https://www.rhino3d.com/`

Building Information Modeling (BIM) tools also provide programming interfaces that enable the use of GD. However it implies a very different way to describe the models using this BIM paradigm [24]. For instance, the model of a wall with a door, in a CAD, can be described in any order since there is no semantic inherent to these objects, however, with BIM the user has to define a wall first and, only then, a door can be created with the previously created wall as a parameter.

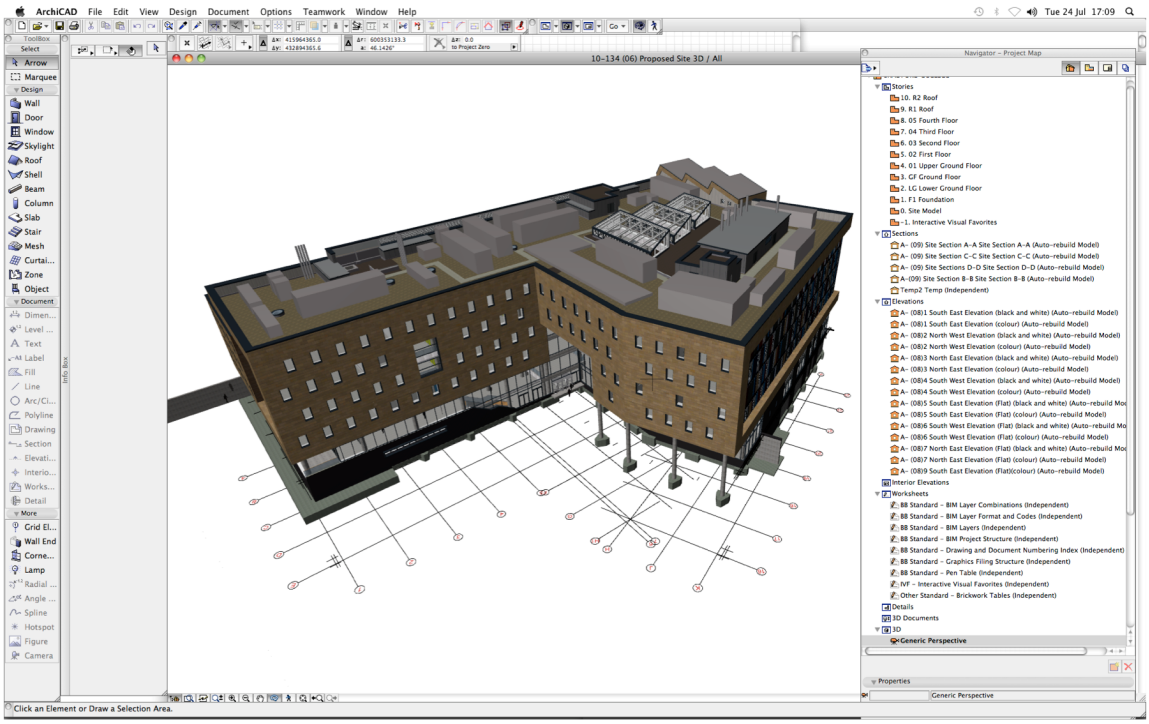

Figure 3.13: Graphicsoft ArchiCad Screenshot

3.8 Rosetta

Rosetta [8][25] is a tool for GD that was created with very specific goals. The focus users are designers and architects, that usually do not have programming experience. So the goals was to be pedagogic, i.e. easy to learn, easy to use. Rosetta explores the CAD tools commonly used as back-ends and also implement various programming languages and front ends. For example, the Figure 3.14 shows the interaction between Rosetta and Rhinoceros.

3.8.1 Architecture

Rosetta implements the architecture described in Figure 3.15. It implements connections with the most used CAD and BIM tools (Section 3.5 & Section 3.7), such as AutoCad[5], Rhinoceros 3D[6], SketchUp[7], ArchiCad[8] and Revit[9]. These tools are the *backends*.

As mentioned before, these tools where not developed for manual use and unfortunately have performance problems when dealing with the large amounts of geometry that GD tools are able to generate. This problem is aggravated with BIM tools since the amount of information that is processed is

[5] `http://www.autodesk.com.br/products/autocad/overview`
[6] `https://www.rhino3d.com/`
[7] `http://www.sketchup.com/`
[8] `http://www.graphisoft.com/archicad/`
[9] `http://www.autodesk.pt/products/revit-family/overview`

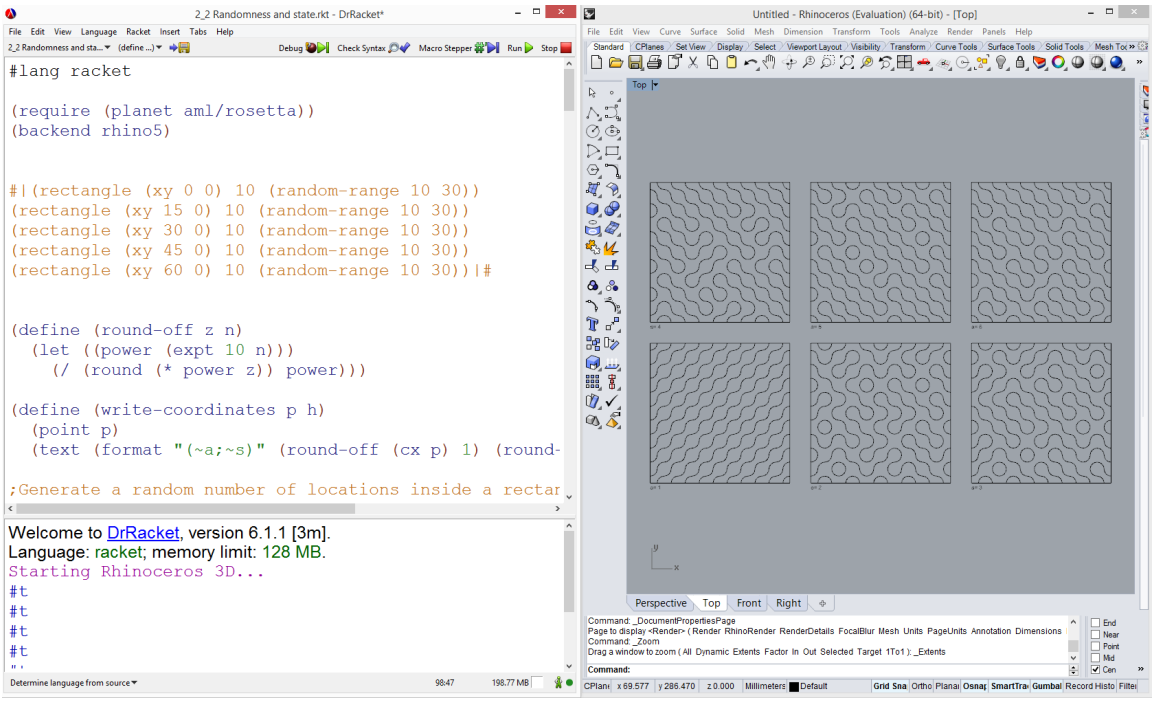

Figure 3.14: Rosetta IDE with Rhino

considerably higher, because, for instance, they have to make comparisons between objects to detect interferences.

Rosetta implements several Programming Languages that the designers can use as *frontends*. The PLs that are currently supported are Racket, AutoLISP, Javascript, Processing, and Python.

3.8.2 Usage

Rosetta is built within the DrRacket IDE [25], and its used through that IDE. As the Figure 3.14 shows, the user chooses a backend with one line of code, that is followed by the description of the model. The backend and the frontend can be any combination of the supported ones. In this case, Racket and Rhino were used as frontend and backend respectively. This results in an highly portable system.

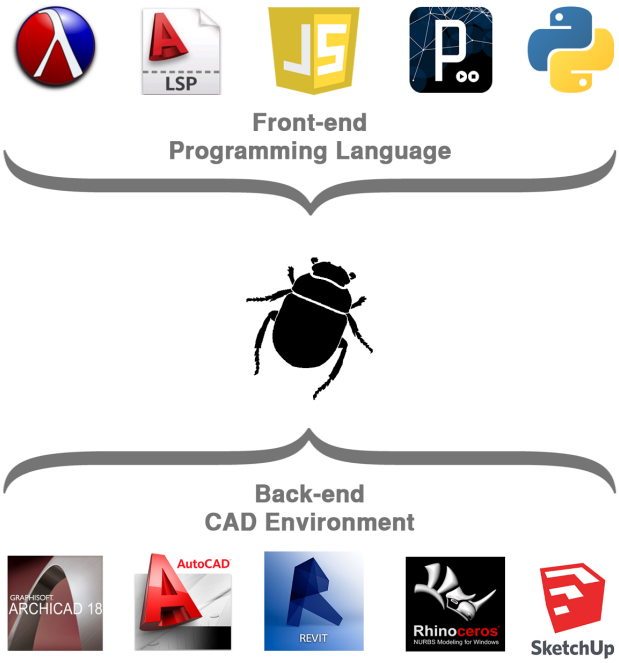

Figure 3.15: Architecture of Rosetta

3.9 Pict3D

https://docs.racket-lang.org/ts-guide/ Pict3D[10] is a tool written in Typed Racket[11] that provides a purely functional interface to rendering hardware. This interface provides a set of geometric primitives that allows the user to produce a large set of very different models. This tool is based on pict[12], one of the standard Racket functional picture libraries.

This tool provides also different visualization mechanisms. It takes advantage of the DrRacket IDE and can show the models in its Read Eval Print Loop (REPL) along with the usual windowed visualization.

The big issue with this tool is performance. DrRacket have known performance issues, because it instruments the code for debug by default and also the garbage collection system is sometimes slow.

It provides a large set of geometric constructors such as, spheres, boxes, etc., and also various operations to manage lights. In the example in Figure 3.16, there are two spheres and an light object that illuminates the scene.

[10] `https://pkg-build.racket-lang.org/doc/pict3d/index.html`
[11] `https://docs.racket-lang.org/ts-guide/`
[12] `https://pkg-build.racket-lang.org/doc/pict/index.html`

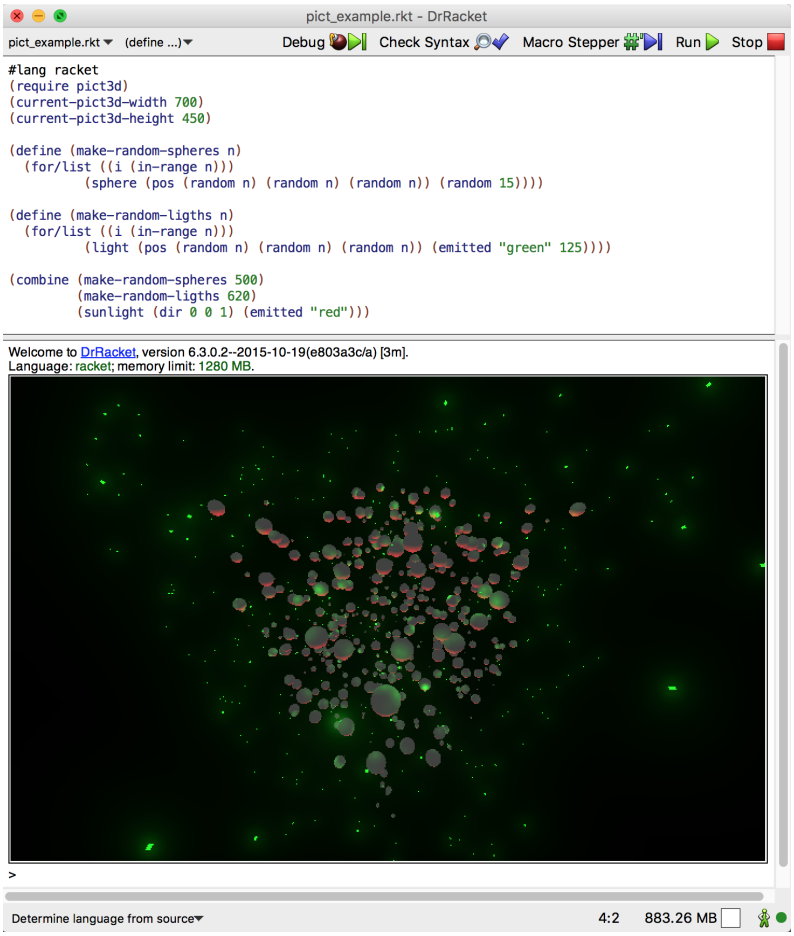

Figure 3.16: Pict3D example within DrRacket

3.10 Related Work Analysis

The previous sections presented a few different systems or projects that are somehow related to ours. The main focus of the first couple of projects, (Section 3.1&Section 3.2) is the visualization of large amounts of geometry in real time. Both use similar techniques to achieve that goal, mainly the ones discussed in Section 2.4.1 and Section 2.4.2. From those systems we learned how we can apply those techniques to improve our system's performance.

The following two projects, *CityEngine* and *No Man's Sky* (Section 3.3 and Section 3.4) are examples of the usage of generative design in different contexts. The first is a general tool to generate city models and can be applied in several different areas, such as simulation, construction industry and in the entertainment industry. The second is an example of an application of generative design to create the game's universe.

From the CAD tools, BIM tools, Grasshopper, and Rosetta (Sections 3.8, 3.6, 3.7 and, 3.8), we found out what primitives were necessary to implement, since they have the same user group that we do.

Pict3D was important since it is written in Racket, which is the same language that we use for front-end, and it is also aimed at learning programmers. However, as it is focused on Racket, it does not support other languages.

Chapter 4

GLFast

This Chapter will present our system, GLFast , which provides a 3D modeling alternative that emphasizes performance. Our goal is to provide a solution that is able to quickly regenerate models, thus allowing the users to experiment different variations of their models, mainly during the creative phase, when changes are frequent.

The first section presents the overall architecture of the system we propose as solution. Within the following sections the system components are explained, detailing obstacles, problems and our solutions to face them.

4.1 Architecture

This section presents the architecture of the system we propose as solution.

The system follows the architecture described in Figure 4.1. It has a *core* module that is implemented in C++ using modern OpenGL and GLSL. This module implements all the core functionality of GLFast . It exports a low level C API. To use this module we propose a higher level *wrapper API*, and with this, allow the use of other languages and, at the same time, simplify the interaction with the tool. We choose to develop one high level API with Racket, and with that we were able to integrate GLFast with Rosetta. To demonstrate the interoperability of our tool, we also developed a simpler version of the Wrapper API in Python (Section 5.1.4).

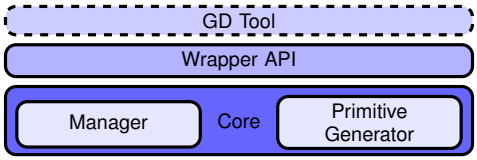

Figure 4.1: Architecture Diagram

4.2 Core

The core of GLFast is written in C++ using modern OpenGL [26]. We use this platform with two main reasons: (1) OpenGL [3] is a cross platform API for graphics, which is important since we want our system to be cross platform, and (2) our previous experience with the platform.

As presented in Figure 4.1, the *Core* has two independent modules, the Manager and the Primitive Generator. The Manager written in C++ has several purposes. It creates and manages the *OpenGL* context and windows, manages the data to be sent to the shaders and implements the rendering loop. The

Primitive Generator contains the shaders, written in GLSL. It is within this module that the primitives are generated, using geometry shaders. The Primitive Generator runs on the GPU, taking advantage of its multi-core architecture to achieve better performance.

4.2.1 Primitive Generator

The most important modules of the *Core* are the *Shaders* since it is there where the primitives are generated. It has simple/pass-through *Vertex* and *Fragment Shaders*. The *Geometry Shaders* are much more complex.

The Geometry Shaders implement the generation of the objects through simple 3D modeling functions. However, since the output type must be specified withing the code, as points, lines or triangles, we could only have one output type for a shader. Our solution is to have different Geometry shaders that deal with different output types. This means that we have a shader that generates solid objects from triangles while lines/wireframes are generated by another shader. This fact allows us to optimize the code to generate solids or wireframes.

4.2.2 Manager

The *Manager* implemented in C++ has several purposes. Its most important role is to create and manage the *OpenGL* context and windows. It also implements part of the system logic, by receiving and processing all the primitive generating calls and creating the data structures to be sent to the *Primitive Generator*.

4.3 Wrapper API

Since the focus users for this system are architects and designers, which do not have large programming experience, several design decisions have been made taking this into account. Because the *Core* API is very low level, we propose a *Wrapper API* to make easier for the users to use our tool through an simpler Programming Language.

The programming language we chose was Racket. Racket is a descendant of Scheme, a language that is well-known for its use in introductory programming courses. Racket comes with DrRacket, a pedagogic Integrated Developlment Environment (IDE) [27][28] used in many schools around the world, as it provides a simple and straightforward interface aimed at inexperienced programmers. As mentioned before, this language also allows the inclusion of GLFast with Rosetta.

The Wrapper API creates a thin layer that implements concepts that are closer to those our users are used to. Concepts, such as ***xyz*** *(a point in 3D)*, are not available within the Core. Currently, it is within the Wrapper API that is implemented the creation of more complex shapes that are composed by simpler ones. Although this is completely transparent to the user, it makes more complex to create other Wrapper APIs with other PLs.

4.4 Implementation Details

This section explains GLFast . It presents some implementation issues that we have faced and the respective solutions that we developed to solve them. As expected, the main problem GLFast aims to solve is about performance. Current GD tools are simply too slow. We attack this problem in a number of fronts, form diminishing the amount of data that is transferred between layers, to the management of the size of the model that is processed.

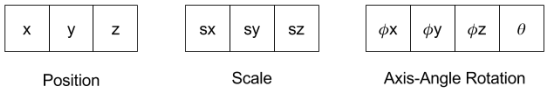

Figure 4.2: Axis-Angle Data Layout

4.4.1 Encoding Primitives Data

We define primitives as any model that can be modeled by one encoding description.

One problem that traditional approaches face is the amount of data that is transferred from the GD tools to the module where it is processed and visualized, usually CAD tools, which is a bottleneck that has impact on the performance of the GD tools.

In order to reduce the amount of data that is transfered between modules, our solution is to transfer the minimum amount of data possible between modules of the system. To achieve this, instead of sending the whole model description, with every point information, such as position, color, etc., a minimum set of descriptors is used to describe each primitive and generate the model as late as possible within the pipeline.

To create any regular primitive, information about three concepts is necessary: position, scale and rotation. Assuming that the objects are created with unitary size, at the origin and oriented along the z axis, it has to be translated, scaled and rotated. Our solutions to store this information are presented below, having as a goal to minimize the data needed.

4.4.1.1 Minimum Data

One solution is to implement the transformation using the minimum possible data.

Translations and scales can be trivially implemented using three floats each, six in total. By centering the object at a desired point, and, since the objects are created with unitary size, by multiplying its size values by the scale.

For the rotation a solution is the axis-angle representation [29], Figure 4.2. This way, only four floats are used, three for the axis of rotation ϕ, and one for the angle θ.

However, this solution is not cumulative, which means that users could not create an object and after that, apply any other transformation without conversions between formats.

4.4.1.2 Transformation Matrix

As an alternative, to solve the mentioned issue, there is another approach which compromises on the used data. The previous solution used a total of ten floats to represent the transformations. By using a 4×4 matrix six more floats are used, but it solves both issues, Figure 4.3. This way we can support cumulative transformations.

For each supported object there is a set of attributes that has to be created. This attributes are an object type, a transformation matrix, the color of the object and a value for the scale, which is used to compute the LOD. Since this approach uses a fixed amount of data that is sent between layers, it might not be optimal for very simple primitives since, some times, they could fit in less data with the traditional approach. But it is easily compensated with more complex primitives. For instance, the data for all the points to model a sphere would be much higher than the data needed using our approach.

4.4.2 Irregular Objects

Irregular objects can not be described with the same amount of information that is needed to describe a regular solid. For instance, is not easy to describe an irregular polygon with our data structure. How-

aa	ab	ac	x
ba	bb	bc	y
ca	cb	cc	z
0	0	0	1

Transformation Matrix

Figure 4.3: Transformation Matrix Layout

ever, it is easy to describe it with a list of its vertices. GLFast uses lists of points to describe *irregular objects*, which removes the need for a transformation matrix. Without the transformation matrix, there is room for 3 points within the fixed data layout. With these 3 points, it is possible to increase the amount of objects that GLFast supports, adding several regular objects such as lines and polygons.

To create irregular polygons with more that 3 vertices, we implemented a triangulation function that, within the CPU, creates the individual primitives that are fed to our shaders.

4.4.3 Primitive Generation

Since we use a description of the models instead of the whole information, the primitives need to be generated at some point. This is done within the GPU with shaders. More specifically using geometry shaders. To generate the primitives in the GPU according to the descriptions, simple modeling parametric algorithms to generate the primitives where implemented. Therefore, the primitives are generated as late as possible, and the generated model is never moved since it is already in the GPU. This fact helps us improve the performance of our system.

Listing 4.1: Shpere Generator Function GLSL

```
void makeSphere(float radius, int sides){
  float leap2 = (PI * 2.0) / float(sides);
  float leap = PI / float(sides);
  vec4 axis = vec4(0.0,1.0,0.0,1.0);
  vec4 camAx = vec4(cameraPos.xy,0.0,1.0);
  float c_angle = dot(camAx,axis)/(length(camAx)*length(axis));
  float angle = acos(c_angle);

  for (int i = 0; i <= sides; i++) {
      float ang = leap * float(i);
      float ang1 = leap * float(i+1);
      // Angle between each side in radians
      for (int j = 0; j <= sides/2; j++)
      {
          float ang2 = leap2 * float(j) + angle;
          float ang3 = leap2 * float(j+1) + angle;

          vec4 offset = MVP * tMat * vec4(sin(ang)*cos(ang2) *
           radius, sin(ang)*sin(ang2) * radius, cos(ang) * radius, 1.0);
```

```
            gl_Position = aux + offset;
            EmitVertex();
            offset = MVP * tMat * vec4(sin(ang1)*cos(ang2) *
             radius, sin(ang1)*sin(ang2) * radius, cos(ang1) * radius, 1.0);
            gl_Position = aux + offset;
            EmitVertex();
        }
        EndPrimitive();
    }
    EndPrimitive();
}
```

GLFast supports a limited set of primitives that we believe are able to produce a meaningful set of different models. Primitives such as Prisms, Pyramids and Spheres are available in the 3D space. However, since all primitives are parametrized, a large set of different models can be generated from these ones. Note that the prisms include Box's, Cylinders and anything in between. GLFast also supports lines and polygons, which can be used as building blocks to create other, more complex, objects.

The Figure 4.4 presents the data Flow of our system. First the user describes the model using a frontend language, in this case Racket. This code through a foreign function call invokes the respective function within our *Core*. There, the data structure to represent the object is created and transfered to the GPU that generates the model.

4.4.4 Model Processing

We aim to support very large models with our system. Thus it is likely that only part of the model is visible from the current camera position, and the rest is outside the viewing frustum. The Graphics Processing Units (GPUs) apply culling techniques to cut out invisible parts of the models when processing the faces. This requires that the entire model is generated before it is cut out. For very large models, this process happens too late. For instance, with a very large model of a city in which only a small portion of it is visible, due to the current camera settings. The whole model is generated and then a large part of it is discarded. This is obviously not good for the performance of the system.

A better solution can be implemented by taking advantage of the timing in which the primitives are generated and applying an Object Culling technique before they are generated, Section 2.4.2. When each description of the object is received by the GPU, the object visibility is computed, taking into account the position of the camera. With this information the generation of the object is prevented, thus generating only the objects that fit on the current view. Figure 2.4 shows a model of a city where only the visible part of the model is generated.

This approach allows another optimization. When an object is very far or is very small, its details are not noticeable. It is usually a waste of resources to have fully detailed objects very far from the camera, when they occupy a very small part of the screen. We implemented a LOD technique (Section 2.4.1) to manage the level of detail that each object is generated with. When each primitive is generated it is trivial to compute how far it is from the camera and the scale it has, and with that compute the level of detail in which it is generated. This means that if an object is very far from the camera, it is generated with less detail without any visible change, thus, taking less time to generate that object and reducing the amount of points to be rendered.

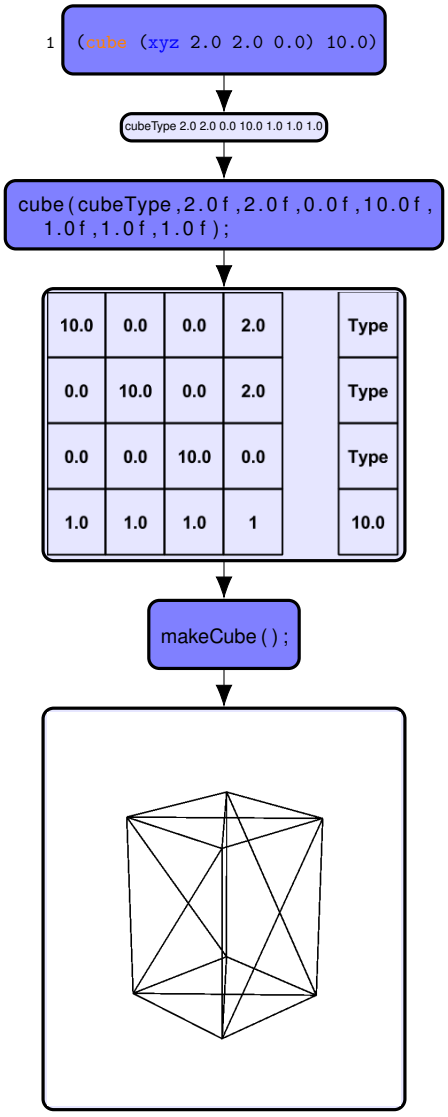

Figure 4.4: Data Flow Diagram

4.4.5 Adding Color

The assignment of color to the objects was a desired feature, to make GLFast more similar to the other systems. One problem about the common approach, which is to simply transport 3 more values, is the growth of the total transported data. A solution to avoid that, is to use the last line of the transformation matrix to transport the color values, since this values are fixed. Thus, adding a totally new feature without adding any more data to transport, just by better using the available space, as shown in Figure 4.5. This trick makes the data usage gap shorter between the option of having the transformation matrix versus the minimum data option, finally resulting in the data layout in Figure 4.6.

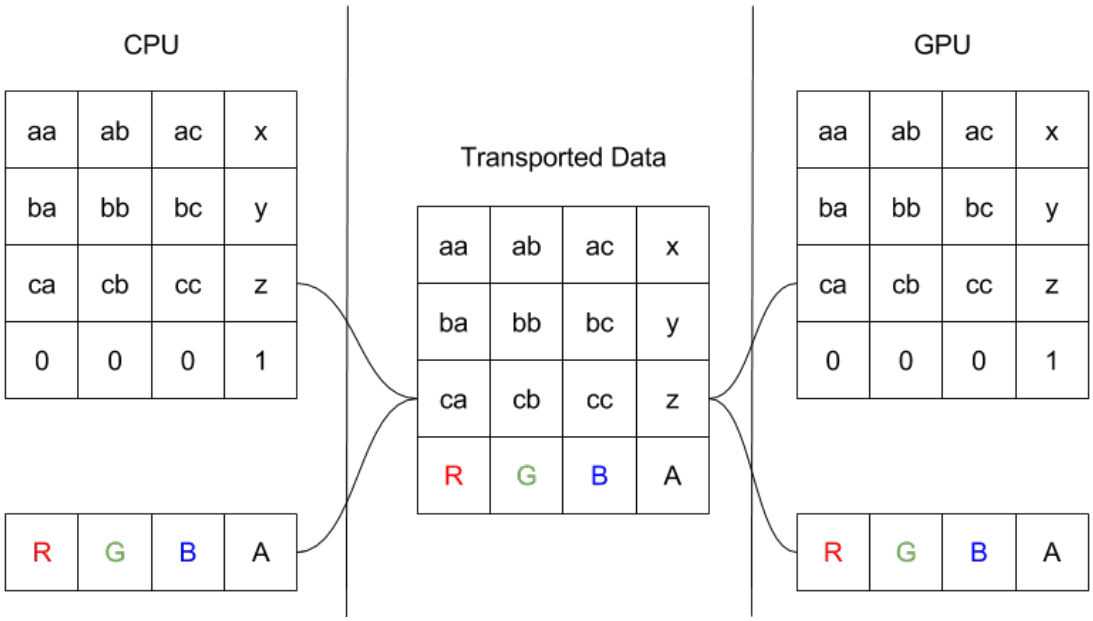

Figure 4.5: Transport of the transformation and color data

20 floats			
matrix (13 floats)	color (3 floats)	type (3 floats)	scale (1 float)

Figure 4.6: Final Data Layout

4.5 Functionality

The following sections present the functionality currently available within GLFast through the *Wrapper API*. First we present the primitives that are supported, which is followed by other operations.

4.5.1 Objects

This section lists, Table 4.1, the primitives that are supported by GLFast . The primitives has an additional argument that defines its color and has white as a default value. The last four are an exception, since they are defined by a type that have to be previously specified and that define all style settings to this objects. Some of these objects are created by composing others, for instance, the polygon is created with triangle mesh. The object type *xyz* is the object that implements a Cartesian coordinate.

The objects beam, column, slab, and roof, are included as a proof of concept for BIM support. The family argument contains all the information needed to construct the object, such as geometry, colors or textures.

Objects Names	Arguments
point	center : xyz
line	pts : List<xyz>
circle	center : xyz , radius : float
reg-polygon	center : xyz , sides : int
reg-polygon-surface	center : xyz , sides : int
polygon	pts : List<xyz>
polygon-surface	pts : List<xyz>
cube	center : xyz , size : float
sphere	center : xyz , size : float
cylinder	base : xyz , top : xyz , radius : float
right-cuboid	base : xyz , top : xyz , width : float, hight : float
prism	base : xyz , top : xyz , width : float, hight : float, sides : int
pyramid	base : xyz , top : xyz , width : float, hight : float
irregular-pyramid	base-pts : List<xyz>, top : xyz
extrusion	base-pts : List<xyz>, hight : float
beam	p1 : xyz , p2 : xyz, family : Family
column	center : xyz , base : level, top : level , family : Family
slab	vertices : List<xyz>, base : level , family : Family
roof	vertices : List<xyz>, base : level , family : Family

Table 4.1: List of the available Primitives

4.5.2 Operations

This section lists, Table 4.2, other operations that are available through GLFast . *View* sets the camera position and to where it looks at, *mirror* creates mirrored copies of the objects according to a point and an axis, *rotate* and *transform* apply this transformations to the primitives, *init* and *start* sets up the library and starts the rendering respectively.

Operation Name	Arguments
view	camera-pos : xyz , look-at : xyz
mirror	object-ids : List<int>, pt : xyz , axis : xyz
rotate	object-id : int , angle : float , axis : xyz
transform	object-id : int, transformation : Mat<float>
init	
start	

Table 4.2: List of the available Operations

4.5.3 Interface

To use the system, we propose the users to use Racket's IDE, DrRacket, to develop the code. Within the visualization window we implement some functions to improve the user's interaction with the models. GLFast has *camera control* with the mouse, using a camera system similar to AutoCAD's.

Our tool also has some shortcuts to change the view of the model. We allow, for instance, to user to change the rendering model between solid, wireframe, points, and solid with edges.

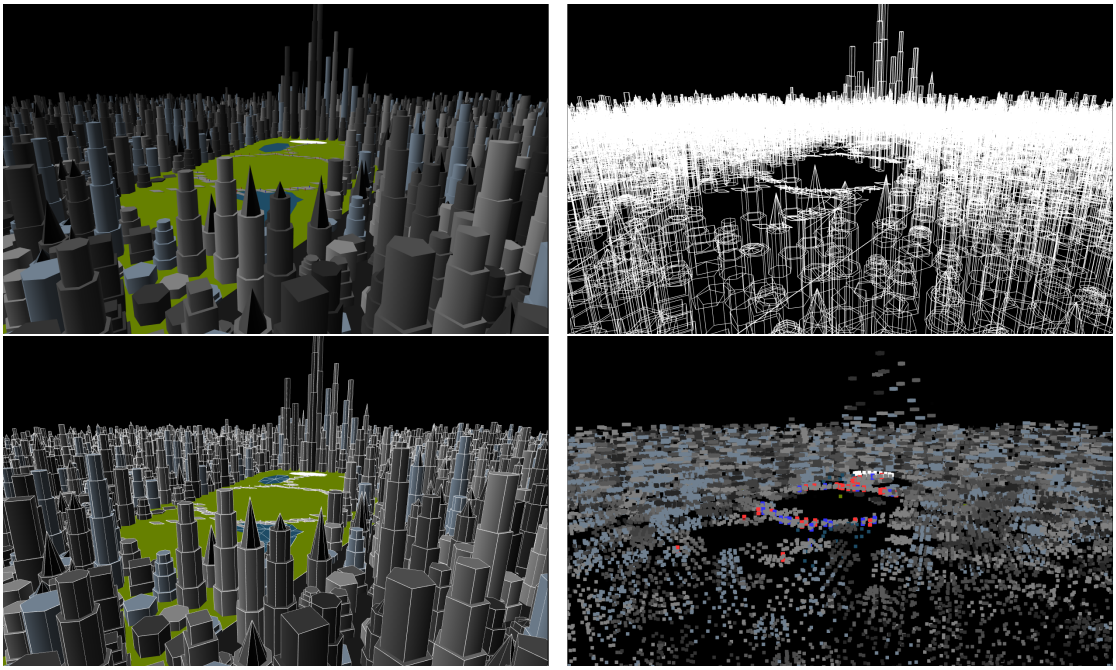

Figure 4.7: Different rendering types

Chapter 5

Evaluation

Within this chapter we will present the evaluation for GLFast .

First we present a functional evaluation that demonstrate the capabilities of GLFast and, after that, we present a performance evaluation, which was done with several benchmarks and compared with results from related systems.

5.1 Functionality

In this section we present some examples of models that were produced using GLFast tool and other tools such as Rhinoceros 3D, AutoCAD for comparison.

5.1.1 Truss Models

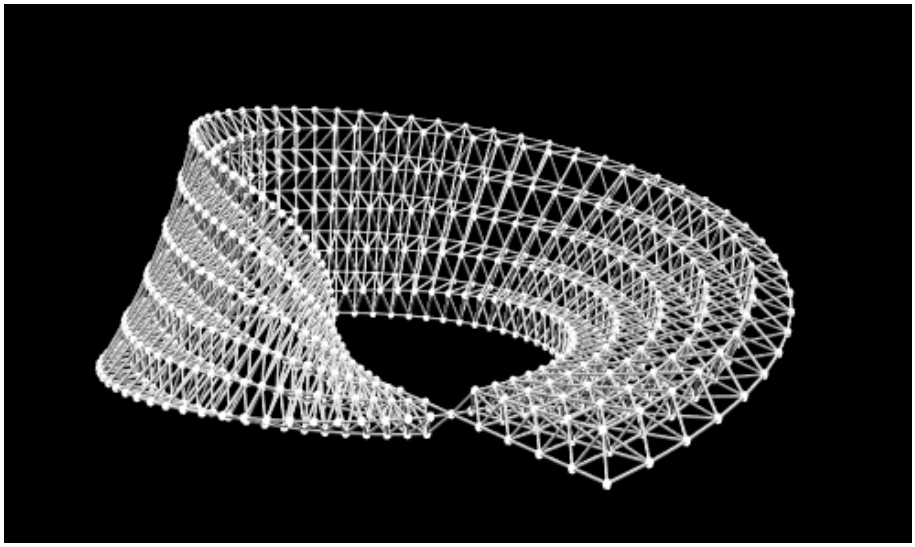

Figure 5.1: Möbius strip Truss Model

Figure 5.1 shows a truss model generated by GLFast . This is an architectural example that is very hard to model manually and would take days or weeks, since each sphere and cylinder have different positions and orientation. It is also very heavy for the CAD tools to generate due to the large amount of geometry, and is generated by the Racket code in Appendix A.1 through Rosetta.

With this code, which is parametric we can generate almost any truss with any given shape. Figures 5.2 and 5.3 are examples of different shapes with a truss.

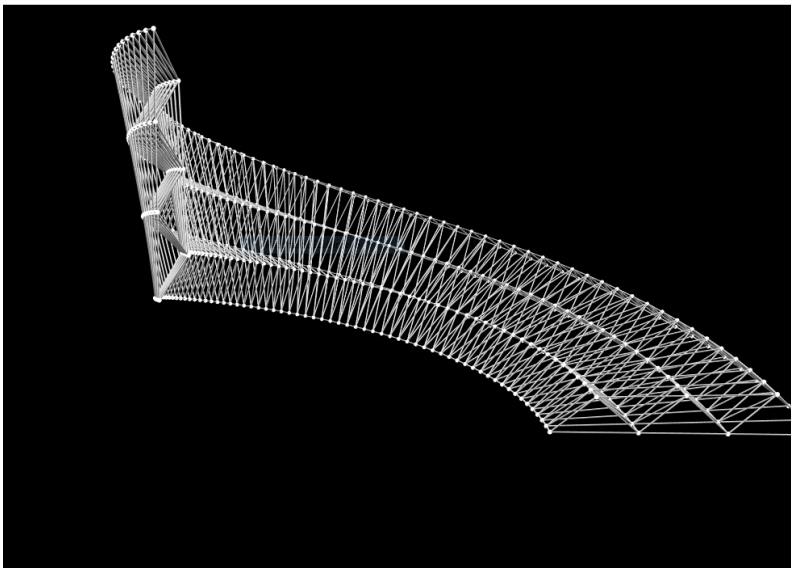

Figure 5.2: Möbius strip shape with Truss

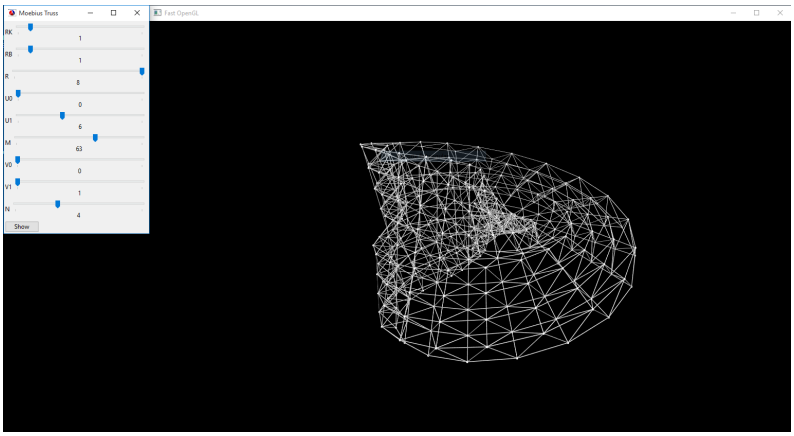

Figure 5.3: Möbius strip variation with Truss and Sliders

5.1.2 Brick Wall

Figure 5.4 presents the wall with bricks that follows a sin curve. This example also shows the sliders that can be used to manipulate the model. In this case, the sliders define the width and height of the wall and the number of bricks in both directions.

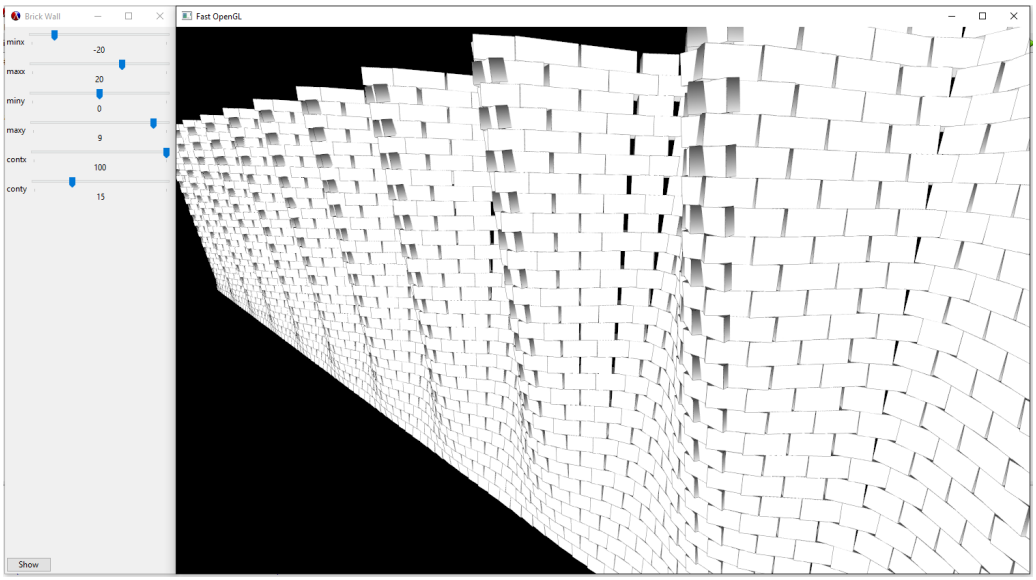

Figure 5.4: SinWall with sliders

5.1.3 Cities

Figure 5.5 presents a model of a city, to show that this tool is able to cope with very large models. In this example, the city contains approximately 160000 buildings with different sizes, shapes and colors. Which represents ∼**7 million polygons**. This example was implemented using C++ calling the Core functionality directly.

5.1.4 Extending GLFast to other programming languages

This section shows how to extend GLFast to be used with other PLs. For this example we use Python, as its recent adoption in Introductory Programming Courses, makes it useful for our users.

The following code is a simple wrapper API for Python which supports only two shapes, *spheres* and *cubes*:

```
from ctypes import*
import random

shading = cdll.LoadLibrary(‘‘glfast.lib’’)

def init(size):
  shading.init(size)

def start():
  shading.start()

def sphere(x,y,z,radius,r,g,b):
  shading.sphere(x,y,z,radius,r,g,b)

def box(x,y,z,sizex,sizey,sizez, r, g, b, angle = c_float(0.0), vx = c_float(0.0), vy = c_float(0.0), vz = c_float(0.0)):
  shading.box(x,y,z,sizex,sizey,sizez, r, g, b, angle, vx, vy, vz)

```

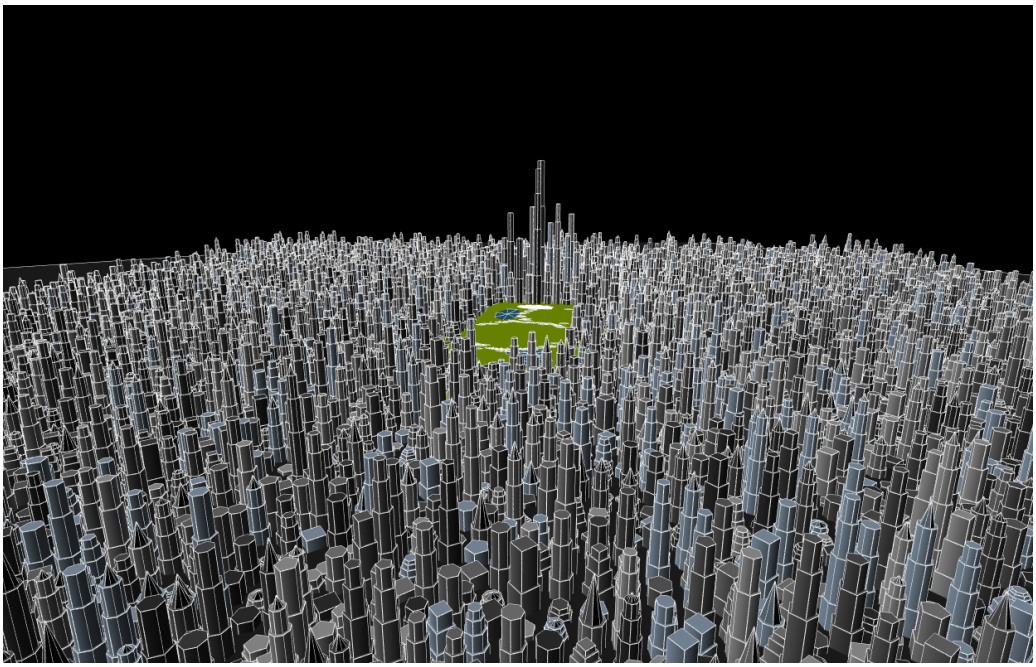

Figure 5.5: City Model using GLFast

```
def cube(x,y,z,size, r, g, b):
  shading.box(x,y,z,size, size, size, r, g, b, angle, vx, vy, vz)
```

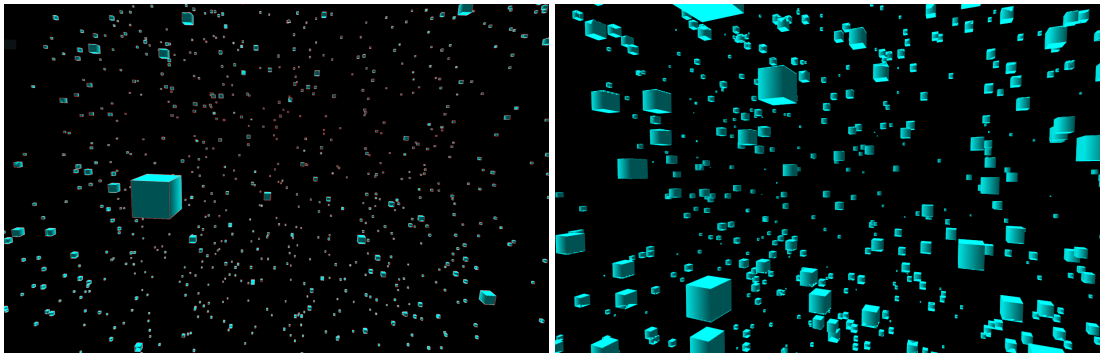

Figure 5.6: Cubes using Python

Despite being very simple, this implementation is functional. The Figure 5.6, which is a set of cubes with aleatory sizes that are randomly distributed throughout a 3D space, was generated by the following code:

```
from ctypes import*
import glfast

f0 = c_float(0.0)
f1 = c_float(1.0)
f3 = c_float(3.0)
f10 = c_float(10)

def randomCubes(n_cubes,world_size):
  for i in range(1,n_cubes):
    size = c_float(random.uniform(2, 15))
    cube(c_float(random.uniform(-world_size, world_size)),
         c_float(random.uniform(-world_size, world_size)),
         c_float(random.uniform(-world_size, world_size)),
         size,
         f0, f1, f1)
```

```
init(1000)

randomCubes(1550,500)

start()
```

5.1.5 Moroccan Pattern

The Figure 5.7 shows a Moroccan Pattern produced using GLFast . This model is a real life example created by an architect for a project he worked on. Figure 5.8 shows the same model but using Rhino.

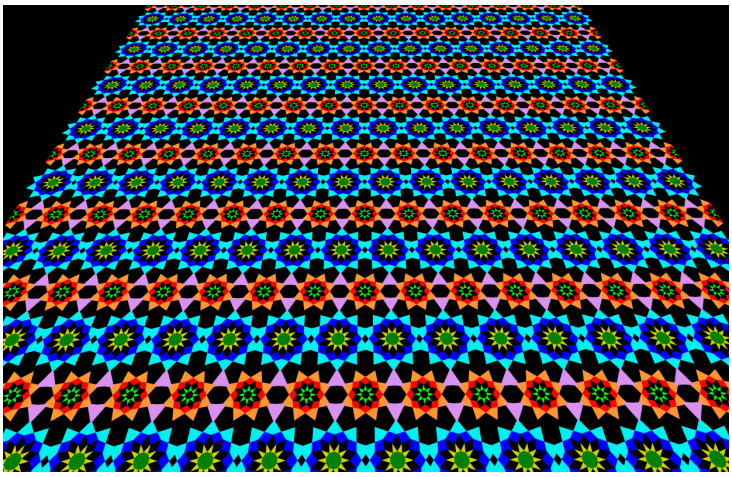

Figure 5.7: Moroccan Pattern from real a project using GLFast

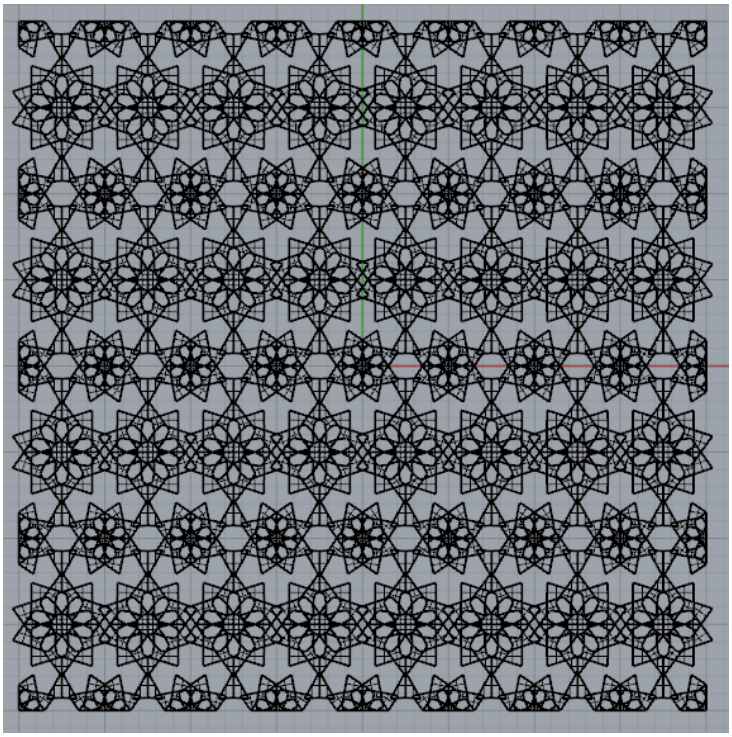

Figure 5.8: Moroccan Pattern from real a project using Rhino

5.2 Performance Evaluation

The following sections will present the results of the performance tests made with GLFast and other related tools. Since the goal is to provide an alternative to the use of CAD tools with Rosetta, several examples were generated using the other backends supported by Rosetta, namely AutoCAD[1] and Rhinoceros 3D[2]. Different tests were made to evaluate different aspects regarding performance. Since the main focus is to show a final result of a model without much interactivity, the usual metrics for graphics performance, namely Frames per Second (FPS) and Frame Time, are not the best. The most important metric is the time that a tool takes to generate the *first image*. However, we used FPS in one test to evaluate the generation of the geometry in the CPU versus GPU.

These benchmarks, and other examples presented throughout this document, were done using a machine with Windows 10, 64-bit, an Intel Core i7-4900MQ CPU @ 2.80GHz, 32 GB RAM, and an NVIDIA Quadro K3000M 2048MB GPU.

5.2.1 Moroccan Pattern Test

The Figure 5.9 shows the results that the several tools had with different configurations of the **Moroccan Pattern**, Section 5.1.5. The chart shows significant gains in performance for GLFast .

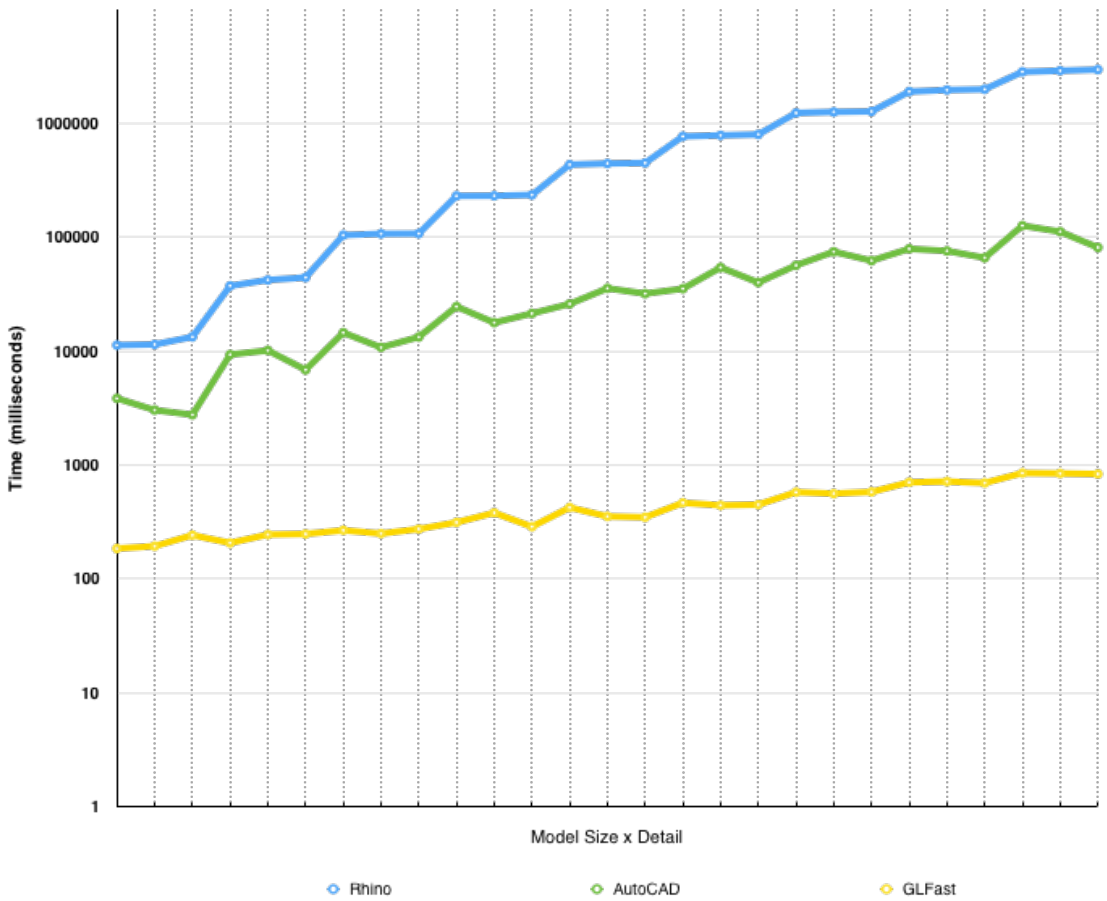

Figure 5.9: Tools performance with different pattern configurations, varying size and detail - Logarithmic Scale

[1] http://www.autodesk.com.br/products/autocad/overview
[2] https://www.rhino3d.com/

5.2.2 Building Test

A different test was made to include the other backends supported by Rosetta, the Figure 5.10 shows the resulting model within all the backends, starting with GLFast at the top left corner, followed, per line, with Rhinoceros 3D[3], ArchiCAD[4], SketchUp[5], AutoCAD[6], and Revit[7]. The Figure 5.11 presents the times that each backend spend to show the final generated model. As we can see, there are major differences between each one of them, and GLFast is much faster than the others.

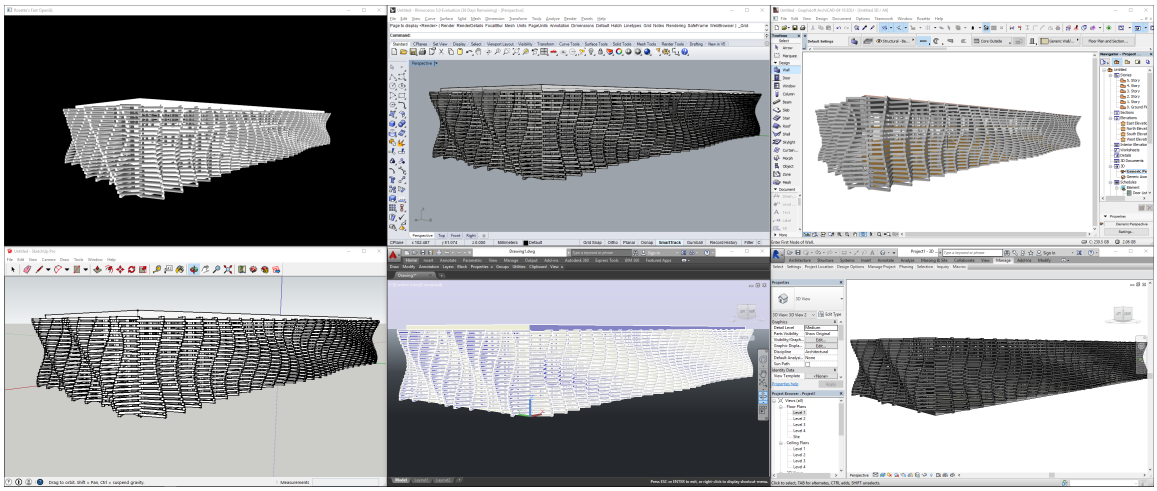

Figure 5.10: Building example results using different backends

5.2.3 GPU vs. CPU Test

Another test was made to evaluate the differences between our approach, using shaders, versus generating the models in the CPU. This was done by implementing a simple backend with OpenGL, using the same algorithms to generate the models.

This test was made using the example of the **truss**, Section 5.1.1, which is a type of model that is very hard to create manually and very heavy, due to the large amount of objects, to visualize with CAD tools. So this is a good example to show the usefulness of our tool. Figure 5.13 shows the results that the tested tools had with different configurations. Figure 5.14 shows the CPU usage between these two options. As mentioned before, with small and simple models GLFast can have lower performances but the gains are significant with larger models.

This test shows that GLFast can present better performance than our simple OpenGL backend.

[3] https://www.rhino3d.com/
[4] http://www.graphisoft.com/archicad/
[5] http://www.sketchup.com/
[6] http://www.autodesk.com.br/products/autocad/overview
[7] http://www.autodesk.pt/products/revit-family/overview

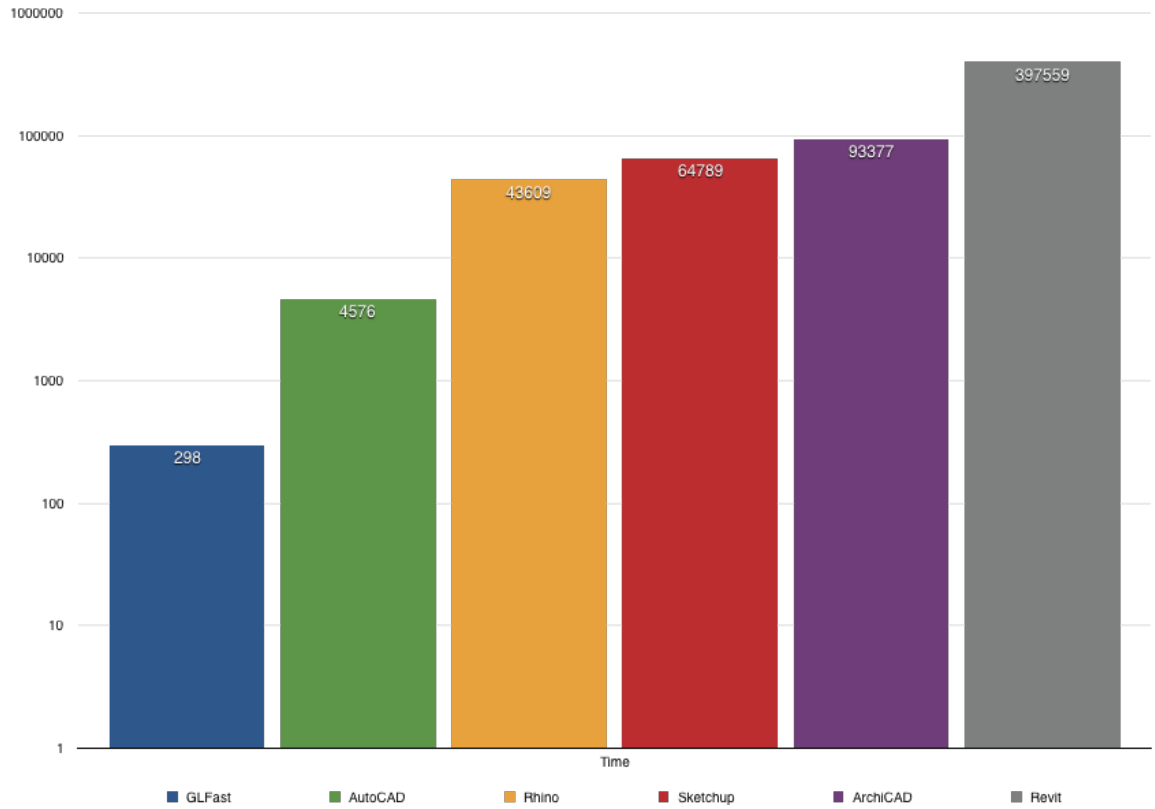

Figure 5.11: Time to generate results using different backends - Logarithmic Scale

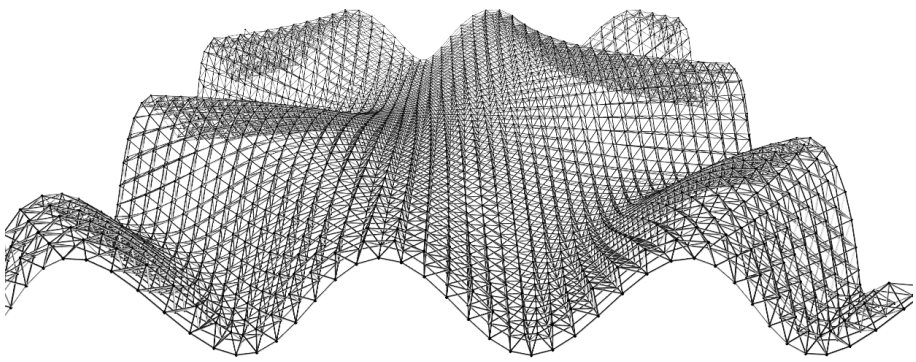

Figure 5.12: Simple Sinusoidal Truss Model

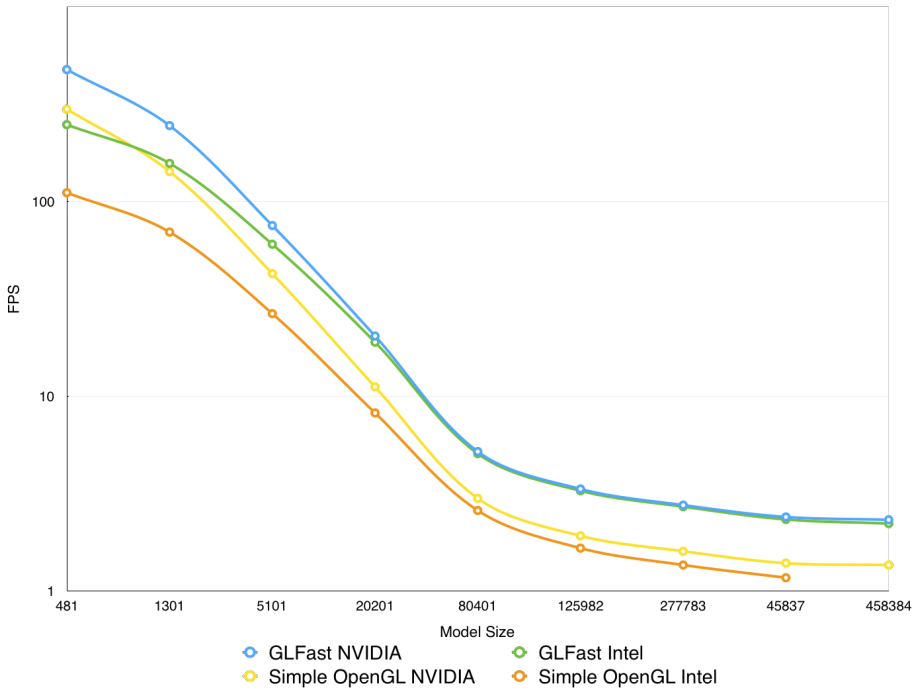

Figure 5.13: Truss FPS test

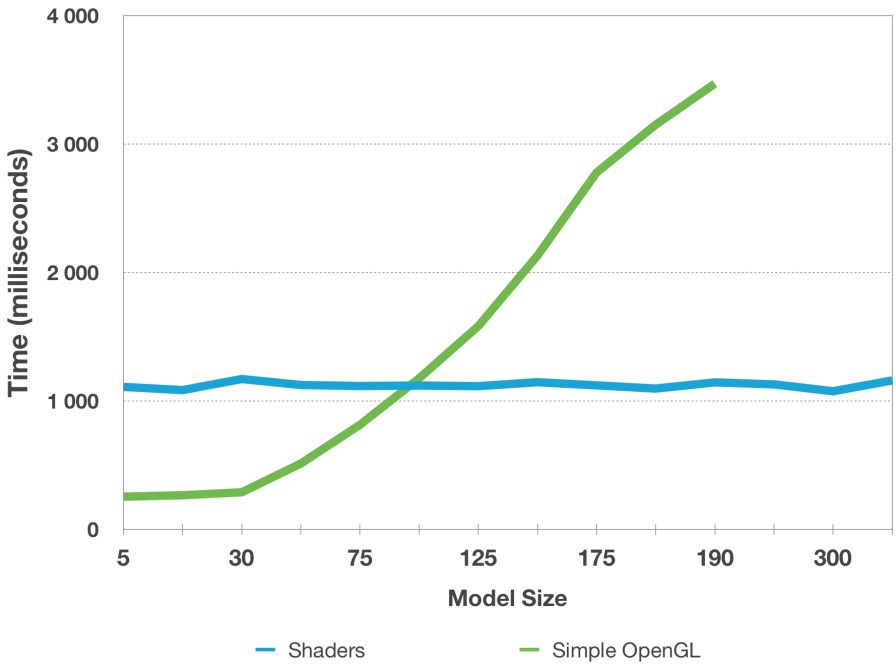

Figure 5.14: CPU usage with and without shaders

Chapter 6

Conclusion

Architects and designers increasingly use programming as a tool. This powerful tool enables them to work faster and with greater creative freedom. With the development of their programming capabilities they begin to create larger and more complex models. This, unfortunately, creates a performance problem since the CAD systems being used were not built for this kind of use. They were developed for a manual, slow usage, because it was not thought at the time that a user could generate massive amounts of geometry in seconds. As a result generative design users have to wait for large periods of time before they can see the results of their programs.

This is a relevant problem which this thesis attempts to solve. To become a valid alternative to the currently used tools, our solution must have good performance and support the most used functions that the users needs.

The defined requirements led to a simple architecture, that has proved to be a right choice, as it provides good performance and, at the same time, allows the extensibility of our tool to work with different programming languages. This resulted in a shorter pipeline than the commonly used.

Regarding implementation, the use of shaders was a risky choice that turned out to be a good one. Also, the other applied techniques, such as Level of Detail Management and Culling, had significant impact to improve performance.

We proposed goals regarding three different aspects for our tool and we analyze each one in the following way:

- **Performance** - GLFast 's current performance met the goals, since it largely reduces the visualization times in comparison to the other tools, achieving very good performance levels, as shown in Chapter 5.2.
- **Portability**
  - Since GLFast 's *Core* was developed in $C/C++$, which is a language known to run in almost any system, it has high portability. Racket also runs on the most used Operative Systems (OSs). This means that our tool can support *Windows*, *MAC OS X* and *GNU/Linux*. It happens that the current thread implementation of Racket for *MAC OS X* and *GNU/Linux* is deprecated, which result in an unstable coordination between the *Wrapper API* and the *Core*.
  - We demonstrated that it is very simple to use our *Core* with other PLs, the Examples 5.1.3 and 5.1.4 in Section 5.1 show that GLFast can be easily used with languages such as C++ and Python.
- **Completeness** - Although our goal was not to replace a CAD tool completely, we still managed to implement some of its functionality, as presented in Section 4.5. The set of examples presented

throughout this documents shows that GLFast is complete enough to create a lot of different models.

We believe that we have been able to achieve the objectives defined in these three aspects. From that we conclude that GLFast is a large benefit to this user community, due to the fact that they have now a different tool to use throughout their creative process.

6.1 Future Work

There are many features that would benefit this project but due to the time frame for this thesis, are left to future work, namely:

- Although the overall performance of the tool is good, as previously shown, some primitives are naively implemented and would benefit from a better implementation. Some primitives are composed by a set of other primitives which implies the increase of the amount of data that is transported. These improvements can be achieved by profiling the system to find out where are the points that are having the largest impacts on performance;
- Investigate if the implementation of independent shaders for each primitive type would improve performance, i.e., whether the elimination of points of decision within the GPU, by choosing which code to run before, within the CPU, and then running it without further tests in the GPU;
- This system would be improved if the set of primitives that supports were expanded, thus increasing the creative freedom of users, including more high level objects such as *building* or *city*;
- Include illumination and shadows to the system, which would improve the rendering quality of the models;
- The implementation of other wrapper API's with different PLs to allow more users to implement their models directly using our library;
- Solve issues regarding the operation in non-Windows OSs, such as the threads issue presented before;
- Explore different APIs for graphic content such as DirectX 12, Vulkan[30] and Metal[1] to see if there are significant improvements in performance.

We believe that our tool and, therefore our users, would benefit from these improvements.

[1] `https://developer.apple.com/metal`

Bibliography

[1] Y. I. H. Parish and P. Müller, "Procedural modeling of cities," *Proceedings of the 28th annual conference on Computer graphics and interactive techniques - SIGGRAPH '01*, pp. 301–308, 2001.

[2] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool, "Procedural modeling of buildings," *ACM SIGGRAPH 2006 Papers on - SIGGRAPH '06*, p. 614, 2006.

[3] D. Shreiner, G. Sellers, J. M. Kessenich, and B. M. Licea-Kane, *OpenGL programming guide: The Official guide to learning OpenGL, version 4.3*. Addison-Wesley, 2013.

[4] D. Shiffman, S. Fry, and Z. Marsh, *The Nature of Code*. D. Shiffman, 2012.

[5] "How to use perlin noise in your games." `http://devmag.org.za/2009/04/25/perlin-noise/`. Accessed: 2015-01-15.

[6] T. Bousfield, *A Practical Guide to AutoCAD AutoLISP*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[7] P. P. Ramos and A. M. Leitão, "Implementing Python for DrRacket," in *3rd Symposium on Languages, Applications and Technologies* (M. J. V. Pereira, J. P. Leal, and A. Simões, eds.), vol. 38 of *OpenAccess Series in Informatics (OASIcs)*, (Dagstuhl, Germany), pp. 127–141, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014.

[8] A. Leitão, "Programação para arquitectura," 2012.

[9] S. Abrishami, J. S. Goulding, F. P. Rahimian, and A. Ganah, "Integration of BIM and generative design to exploit AEC conceptual design innovation," *Journal of Information Technology in Construction*, vol. 19, pp. 350–359, 2014.

[10] S. Krish, "A practical generative design method," *CAD Computer Aided Design*, vol. 43, no. 1, pp. 88–100, 2011.

[11] G. Kelly and H. McCabe, "A survey of procedural techniques for city generation," *ITB Journal*, pp. 87–130, 2006.

[12] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-time rendering*. CRC Press, 2008.

[13] "Culling explained." `http://docs.cryengine.com/display/SDKDOC4/Culling+Explained`. Accessed: 2016-2-15.

[14] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley, *Texturing and modeling: a procedural approach*. 2002.

[15] B. B. Mandelbrot, D. E. Passoja, and A. J. Paullay, "Fractal character of fracture surfaces of metals," 1984.

[16] E. W. Weisstein, "Cellular automaton." From MathWorld–A Wolfram Web Resource.`http://mathworld.wolfram.com/CellularAutomaton.html`. Accessed: 2015-05-16.

[17] G. K. Francis, H. Abelson, and A. DiSessa, *Turtle Geometry. The Computer as a Medium for Exploring Mathematics.*, vol. 90. 1983.

[18] G. Stiny and J. Gips, "Shape grammars and the generative specification of painting and sculpture," *Information Processing 71 Proceedings of the IFIP Congress 1971. Volume 2*, vol. 71, pp. 1460–1465, 1972.

[19] G. Kelly and H. McCABE, "An interactive system for procedural city generation," *Institute of Technology Blanchardstown*, 2008.

[20] "Perlin noise." `http://freespace.virgin.net/hugo.elias/models/m_perlin.htm`. Accessed: 2014-10-16.

[21] T. Fukuda, H. Ban, K. Yagi, and J. Nishiie, "Development of high-definition virtual reality for historical architectural and urban digital reconstruction: A case study of azuchi castle and old castle town in 1581," in *Computer-Aided Architectural Design Futures. The Next City - New Technologies and the Future of the Built Environment* (G. Celani, D. M. Sperling, and J. M. S. Franco, eds.), vol. 527 of *Communications in Computer and Information Science*, pp. 75–89, Springer Berlin Heidelberg, 2015.

[22] S. Greuter, J. Parker, N. Stewart, and G. Leach, "Real-time procedural generation of 'pseudo infinite'cities," *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pp. 87–95, 2003.

[23] S. Azhar, "Building Information Modeling (BIM): Trends, Benefits, Risks, and Challenges for the AEC Industry," *Leadership and Management in Engineering*, vol. 11, no. 3, pp. 241–252, 2011.

[24] B. Ferreira and A. Leitão, "Generative Design for Building Information Modeling," *Real Time - Proceedings of the 33rd eCAADe Conference*, vol. 1, pp. 635–644, 2015.

[25] J. Lopes and A. Leitão, "Portable generative design for CAD applications," *Integration Through Computation - Proceedings of the 31st Annual Conference of the Association for Computer Aided Design in Architecture, ACADIA 2011*, pp. 196–203, 2011.

[26] M. J. Kilgard and K. Akeley, "Modern opengl: Its design and evolution," in *ACM SIGGRAPH ASIA 2008 Courses*, SIGGRAPH Asia '08, (New York, NY, USA), pp. 13:1–13:31, ACM, 2008.

[27] R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen, "DrScheme: A Pedagogic Programming Environment for Scheme," *Programming Languages Implementations Logics and Programs*, vol. 1292, pp. 369–388, 1997.

[28] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen, "DrScheme: A Programming Environment for Scheme," *J. Funct. Program.*, vol. 12, no. 2, pp. 159–182, 2002.

[29] T. L. Curtright, D. B. Fairlie, and C. K. Zachos, "A compact formula for rotations as spin matrix polynomials," *Symmetry, Integrability and Geometry: Methods and Applications (SIGMA)*, vol. 10, 2014.

[30] J. A. Shiraef, "An exploratory study of high performance graphics application programming interfaces," 2016.

Appendix A

Appendix chapter

A.1 Truss Code

```
(require rosetta)
(backend glfast)
(define truss-node-radius
  (make-parameter 0.05))

(define (node-truss p)
  (sphere p (truss-node-radius)))

(define truss-radius-bar
  (make-parameter 0.01))

(define (bar-truss p0 p1)
  (when (not (=c? p0 p1))
    (cylinder p0 (truss-radius-bar) p1)))

(define (nodes-truss ps)
  (map node-truss ps))

(define (bars-truss ps qs)
  (for/list ((p (in-list ps))
             (q (in-list qs)))
    (bar-truss p q)))

(define (spacial-truss curves)
  (let ((as (car curves))
        (bs (cadr curves))
        (cs (caddr curves)))
    (nodes-truss as)
    (nodes-truss bs)
    (bars-truss as cs)
    (bars-truss bs as)
```

```
    (bars-truss bs cs)
    (bars-truss bs (cdr as))
    (bars-truss bs (cdr cs))
    (bars-truss (cdr as) as)
    (bars-truss (cdr bs) bs)
    (if (null? (cdddr curves))
        (begin
          (nodes-truss cs)
          (bars-truss (cdr cs) cs))
        (begin
          (bars-truss bs (cadddr curves))
          (spacial-truss (cddr curves))))))

(define (insert-pyramid-vertex ptss)
  (if (null? (cdr ptss))
      ptss
      (cons
       (car ptss)
       (cons (insert-pyramid-vertex-2
          (car ptss) (cadr ptss))
             (insert-pyramid-vertex (cdr ptss))))))

(define (insert-pyramid-vertex-2 pts0 pts1)
  (cons (quadrangular-pyramid-vertex
    (car pts0) (car pts1)
    (cadr pts1) (cadr pts0))
        (if (null? (cddr pts0))
            (list)
            (insert-pyramid-vertex-2
              (cdr pts0) (cdr pts1)))))

(define (render-truss matrix)
  (let* ((p0 (caar matrix))
         (p1 (caadr matrix))
         (p2 (cadadr matrix))
         (p3 (cadar matrix))
         (d (min (distance p0 p1) (distance p0 p3))))
    (parameterize ((truss-node-radius (/ d 9.0))
                   (truss-radius-bar (/ d 19.0)))
      (spacial-truss
       (insert-pyramid-vertex
        matrix)))))

(define (sin-u*v n)
  (map-division
   (λ (u v)
     (xyz (* u 10)
```

```
          (* v 10)
          (* 4 (sin (* u v)))))
    (* -1 pi) (* 1 pi) n
    (* -1 pi) (* 1 pi) n))

(render-truss (sin-u*v 100))
```

Acronyms

API Application Programming Interface. 6, 29, 38, 56

BIM Building Information Modeling. 31, 42

CAD Computer Aided Design. iii, 3, 5, 29–31, 49, 51

Culling Culling. 10

FPS Frames per Second. 49

GD Generative Design. 3–6, 29, 31, 38

GPU Graphics Processing Unit. 40, 41, 56

HD High-Definition. 23

L-System Lindenmayer System. 16

LOD Level of Detail. xiii, 11, 39, 41

OS Operative System. 55, 56

PL Programming Language. 6, 29, 32, 38, 47, 55, 56

REPL Read Eval Print Loop. 33

VR Virtual Reality. 23