Procedural Generation for Architecture

Artur Alkaim
arturalkaim@tecnico.ulisboa.pt

Instituto Superior Técnico, Universidade de Lisboa

Abstract. Existing tools used for 3D modeling creation are mostly geared for manual use. Unfortunately, the manual production of large amounts geometry is very time consuming. Procedural generation of these forms is one of the approaches which considerably speeds up this process. This approach consists in the algorithmic construction of these forms and allows the quick creation of massive amounts of geometry. As most 3D modeling tools were not made specifically for this type of use, instead favoring manual use, they do not have the performance necessary for a smooth use. This work proposes solutions to this performance problem, through the use of different techniques that accelerate the production and visualization of large volumes of geometry.

Keywords:
3D modeling, OpenGL, Generative Design, Procedural Generation, Shaders

1 Introduction

Graphic contents are mainly used for entertainment, both in the gaming and movie industries, but they are 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. The obvious 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, because 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). This is a design method that is based on a programming approach which 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.

Although most computer-aided design (CAD) applications provide programming languages for generative design, programs written in these languages have very limited portability. Additionally, the provided languages, such as AutoLisp, C++ or Visual Basic, are not pedagogical and are difficult to use even to experienced programmers. All this problems create barriers to the adherence to this approach by all users, specially those that are not used to code.[12]

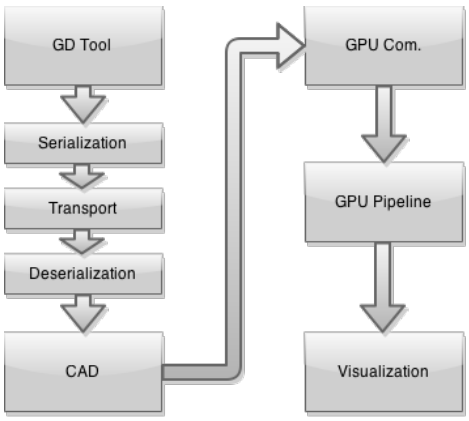


Fig. 1: Common Generative Design Pipeline

There are several generative design (GD) tools such as Grasshopper[1] and Rosetta[8], that aim to break down some of this barriers, and facilitate the approximation of these individuals to programming. With this tools the users can create their models using pedagogical and easy to use languages. This systems implement a straightforward pipeline presented in Figure 1.

Architects and designers implement their models on GD tools. Users implement their models through the GD tool interface. Then all the geometry data is serialized and the data is transfered through some transport mechanism. This data has to be deserialized on the other side within the CAD application. The CAD application takes the deserializes 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. This creates a performance problem.

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. 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 values on their program and see the effects they have on the model. To help them with this, 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.

However, there is a problem: CAD applications are built for manual modeling mainly, and are not prepared to quickly handle large amounts of geometry. Running the code produces much more geometry and much faster than manual

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

modeling, so the user is able to create massive amounts of geometry, which is fed to the CAD that gets overloaded. With this issues, it is hard to get good performance, specially with large models, that makes impossible to have true immediate feedback, which mainly requires visualization, that is only one of the concerns of a CAD application. Because of that users sometimes have to wait long periods of time for the model to be rendered.

This work proposes a solution to this problem and aim 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 implement the generation of the geometry, procedural techniques such as: Fractals(Section 3.3.1), Cellular Automata (Section 3.3.2), and L-Systems (Section 3.3.3) will be applied. To improve performance with visualization, techniques such as Level Of Detail (Section 3.4.1) and Occlusion Culling (Section 3.4.2) are explored.

2 Overview

This section will provide an overview of this topic. Section 4 will address the objectives for this thesis work. Section 3 will explore different yet related work. Section 5 will describe the architecture of the proposed solution. Section 6 will explain how this solution will be evaluated and section 7 will conclude this work.

2.1 Procedural Modeling

Procedural Modeling 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.

The key property of procedural generation is that it describes the data entity, such texture, geometry, or sound, in terms of a sequence of instructions rather than as a static block of data [7]. 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.

3 Related Work

In this section I give an overview of the related work that has been carried out on procedural generation of large amounts of geometry . After that, in the

following sections, some techniques of procedural modeling are explained. Since the target audience of this work are architects and designers, most of the work that I present have the same target.

3.1 CityEngine [11] [10]

CityEngine 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 that are assigned to various indicators in the model that can be as high level as the *size of the city* or as specific as the width of the windows 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.

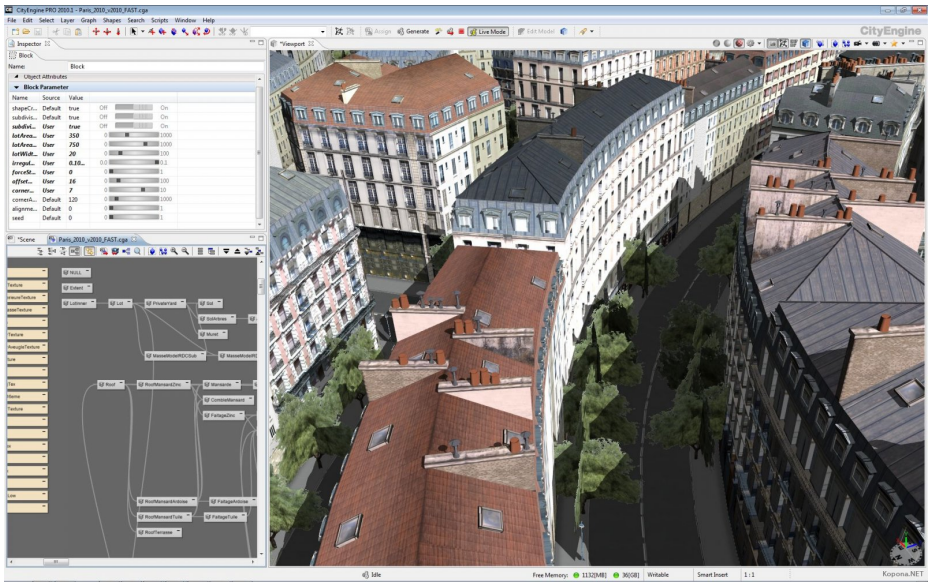


Fig. 2: City Engine Interface

3.1.1 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, but it can be set by the user to a different one.

To implement this growth process, it uses an L-System (Section 3.3.3) that computes the road network.

Fig. 3: Road Map growth

The Figure 3 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.

3.1.2 Buildings To implement the generation of buildings, CGA was created, which is a shape grammar(Section 3.3.4) that was introduced in [11]. 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 an 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 2 shows a set of rules (bottom left), that is relatively small but is already difficult to follow the connections between rules.

3.1.2.1 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 4.

Fig. 4: Basic shapes

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 5.

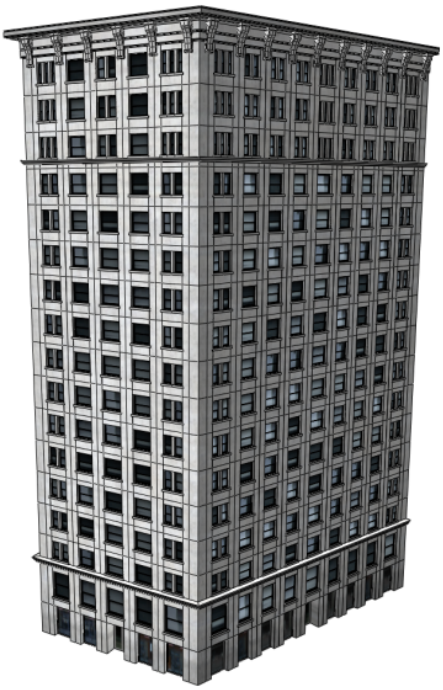


Fig. 5: Complex building modeled with CGA

3.1.3 Cities The result can be a city like Figure 6, with approximately 26000 buildings.

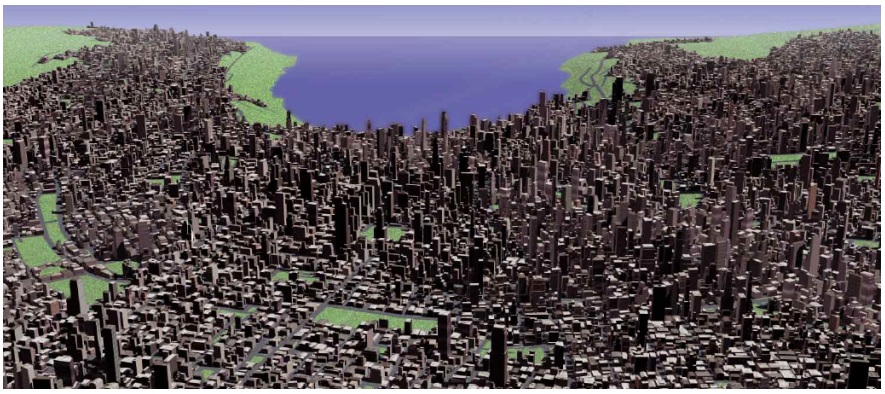


Fig. 6: City with approximately 26000 buildings.

City Engine outputs can be imported by Maya[2], to achieve better results. Like the Figure 7, that represents a 'virtual' Manhattan.

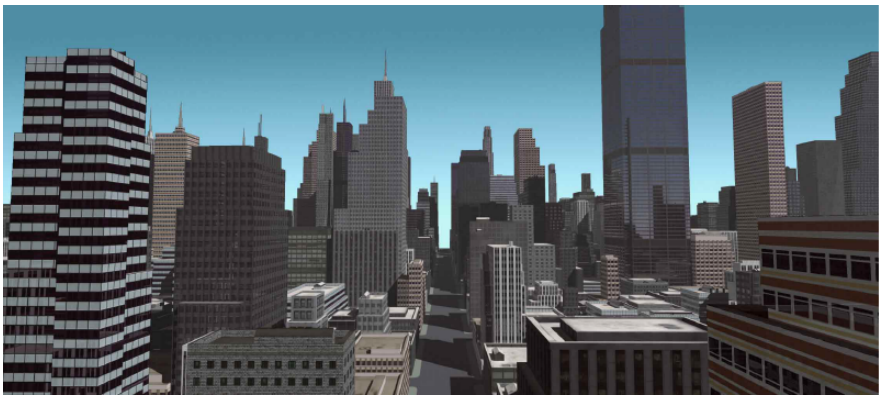


Fig. 7: City rendered with Maya.

3.2 Undiscovered City

In [5] 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

[2] `http://www.autodesk.com/products/maya/overview`

as they are encountered by the user." As shown in the Figure 8 only the part of city that is inside the viewing range is generate. This method allows the visualization of massive amounts of geometry, buildings in this case, by generating in real time only the geometry that on sight, and since this subset is usually much smaller than all the geometry this results in huge benefits in performance.

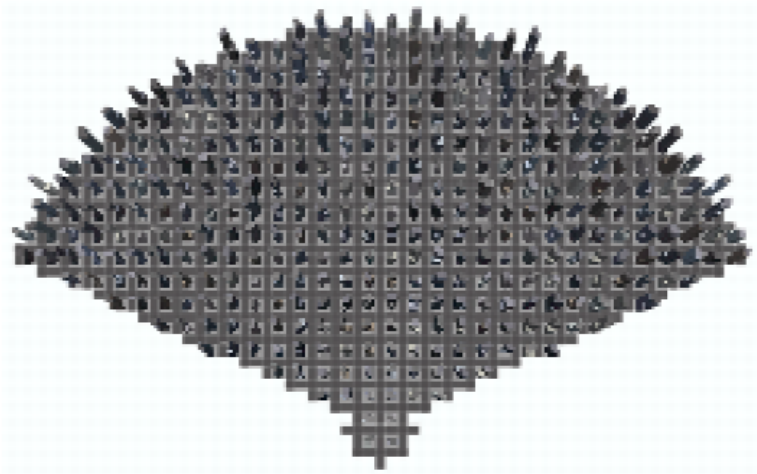


Fig. 8: Viewing Range

3.2.1 Road Network The system uses a 2D grid that divide the terrain into square cells. The cells represent proxies for the content that will be procedurally generated. Before the content of each cell is generated, the potential visibility of it is tested, and 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 very natural, and in the continuation of the work, this system evolved into a more realistic one with the join of some of the grids to create a less uniform distribution of the buildings.

3.2.2 Buildings To compute the form and appearance of each building, it is used a "single 32 bit pseudo random number generator seed. The random sequence determines building properties such as width, height and number of floors." Similar sequences of number result in similar buildings. To avoid that, it is used a a hash function to convert each cell position into a seed.

To generate a building the first is to generate a floor plan. To do so, it is randomly selected and merged a set of regular polygons and rectangles, then this is extruded. This is an iterative process, that creates sections from the top to the bottom, by adding more shapes to the the initial shape and extruding

as shown in the Figure 9. Starting from the left, first there is a simple polygon, that is merged with a rectangle and after extrusion, forms the first block that will be the top of the building. After that, another extrusion is made to generate the next block followed by the merge of a rectangle to the floor shape and the generation of a new block and so on.

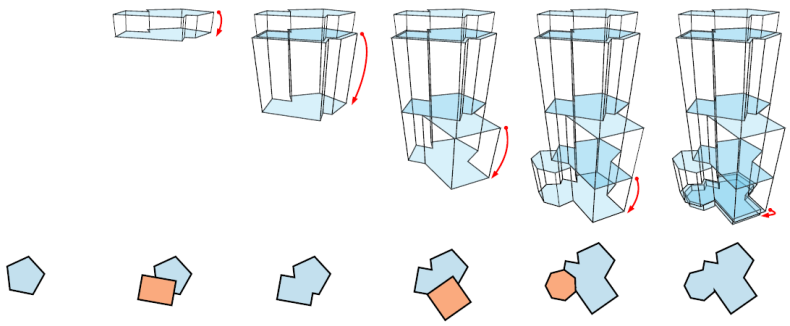


Fig. 9: buildings

With the application of this method very complex architectural forms can be generated, depending only on which forms are selected and the order that is used to merge them.

3.3 Procedural Modeling Techniques

In the following sections are explained some procedural modeling techniques. These techniques are applied to the generation of various types of forms procedurally and will be explored for application within this work.

3.3.1 Fractals A fractal is defined in [4] as "a geometrically complex object, the complexity of which arises through the repetition of a given form over a range of scales". 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 [9]. Fractals were proposed to be seen as a new form of symmetry [4], *Dilation Symmetry*, which is when an object is invariant over a change of scale. This invariance might be only qualitatively 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 11.

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

Fig. 11: Fractals in Nature

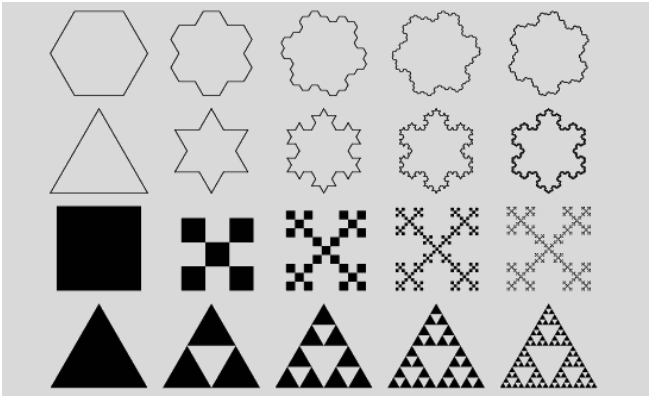


Fig. 12: Geometric Fractals

In Figure 12 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, at 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*"[4]. 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 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.

3.3.2 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 each 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 18), textures, and as inspiration to architecture (Figure 14).

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. This rules are referred by their *Wolfram code* [15].

A common initial state for this elementary cellular automata is a random line. But 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 15 (the rule 30), we get the pattern in the Figure 16 that represents the evolution of a cellular automaton over a few generations.

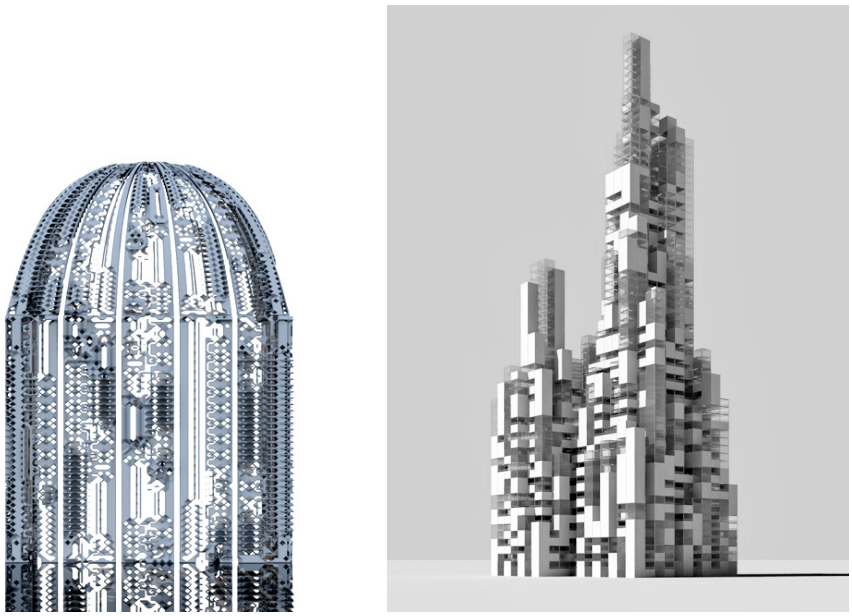


Fig. 14: Examples of cellular automata applied in architecture

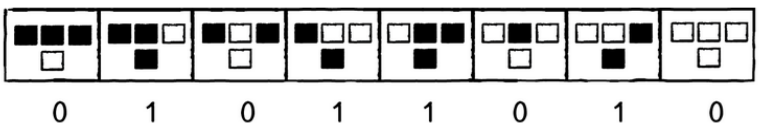


Fig. 15: Example Production Rules[13]

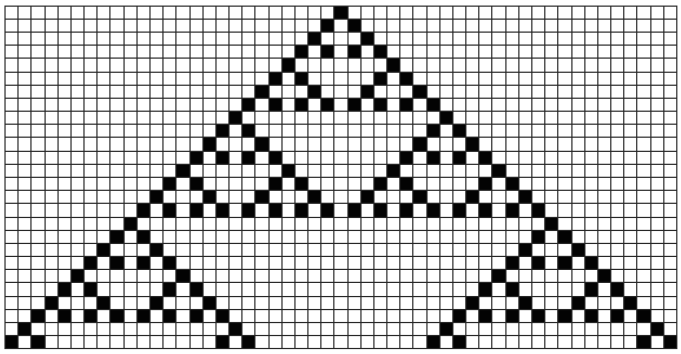
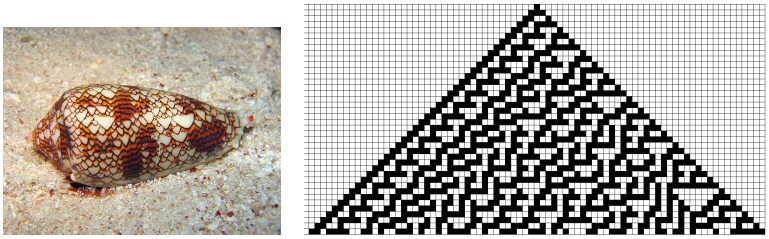


Fig. 16: Sierpiński Triangle, rule 90

In Figure 16 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.

In this context, cellular automata are used to model natural shapes and textures, the Figure 17a shows a natural texture on a Textile Cone Snail that looks like the patterns formed with the cellular automaton in the Figure 17b.



a) b)

Fig. 18: Example of the representation of natural patterns with cellular automata. a) Natural Shell, image from [13]. b) Pattern formed with the rule 30.

3.3.3 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-Systems 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 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 19), and the generation of Fractals.

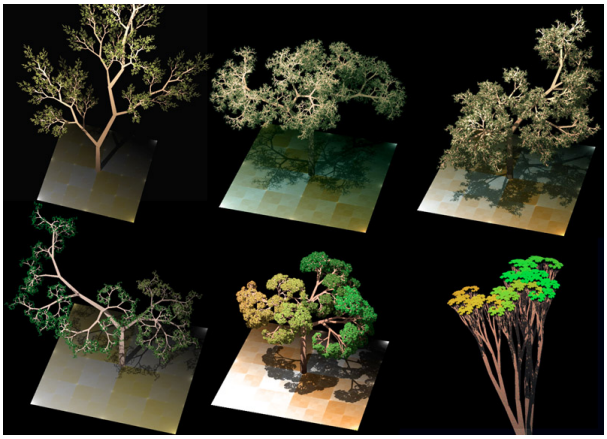


Fig. 19: Trees with L-Systems

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:

$$F \tag{1}$$

$$F + F - -F + F \tag{2}$$

$$F+F--F+F \ + \ F+F--F+F \ - \ - \ F+F--F+F \ + \ F+F--F+F \tag{3}$$

This is an example of the evolution of one system where the production is applied in (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 [3], 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 20.

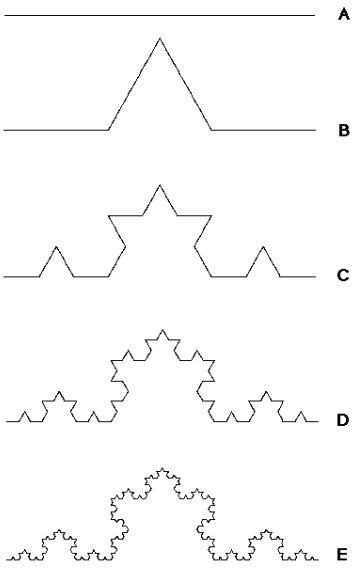
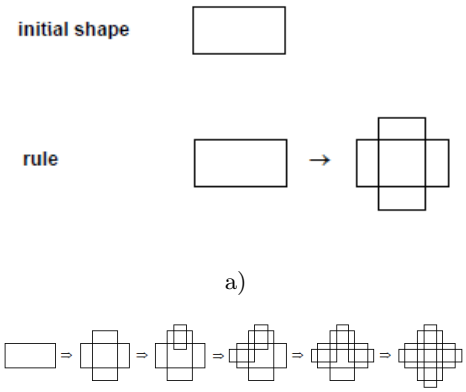


Fig. 20: Result of the “turtle walk” with the given example

3.3.4 Shape Grammars Shape Grammars 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 22 exemplifies one shape grammar with one rule, and the evolution of the application of this rule to the shapes iteratively. In this image, it is shown that from very simple initial shape, a complex from can be generated after a few iterations.



a)

b)

Fig. 22: a) Grammar Tiles b) Recursion steps

In the CityEngine [10] system, this is applied to the generation of buildings using 3D blocks for the main form, and 2D shapes to design the facades.

Figure 23 shows a simple building that I modelled using CityEngine and its CGA Shape Grammar (Section 3.1). But CGA is powerful enough to model much more complex buildings like the one in Figure 24.

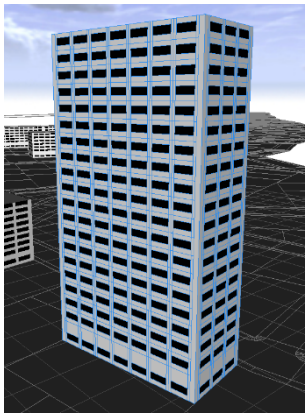


Fig. 23: Simple Building

Fig. 24: Complex Building [10]

3.3.5 Noise "To generate irregular procedural textures, we need an irregular primitive function, usually called noise" [4]. It is a pseudorandom function that breaks the monotony of a pattern and make it look more random. Perlin Noise is the most 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 functions so that the same input returns always the same output sequence.

The properties of an ideal *noise* functions are as follows [4]:

- *noise* is a repeatable pseudorandom function of its inputs
- *noise* has a known range, namely, from -1 to 1.
- *noise* is band-limited, with a maximum frequency of about 1.
- *noise* does not exhibit obvious periodicities or regular patterns. Such pseudorandom functions are always periodic, but the period can be made very long and therefore the periodicity is not conspicuous.
- *noise* is *stationary* - that is, its statistical character should be translationally invariant
- *noise* is *isotropic* - that is, its statistical character should be rotationally invariant

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 forming a coherent noise. These layers are called *Octaves* and the ratio between amplitude and frequency of the layers can be expressed as a constant known as *persistence* [6]. With the result we can create a texture that looks natural and with fractal like structure.

For instance, the Figure 25 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 26 [2].

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 28. 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 29 and Figure 30 shows two cities which 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

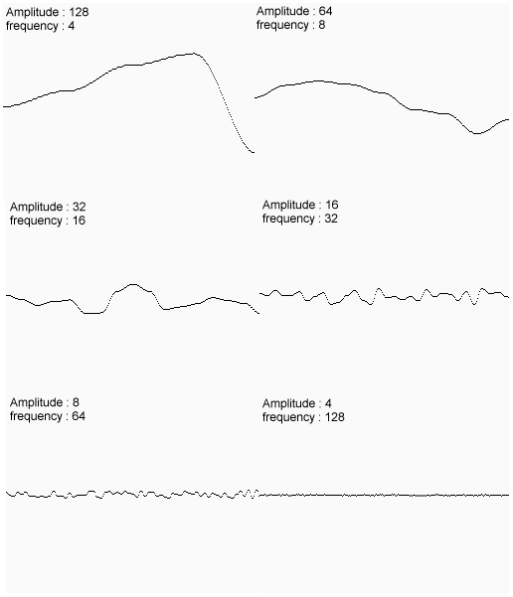


Fig. 25: Different noise functions

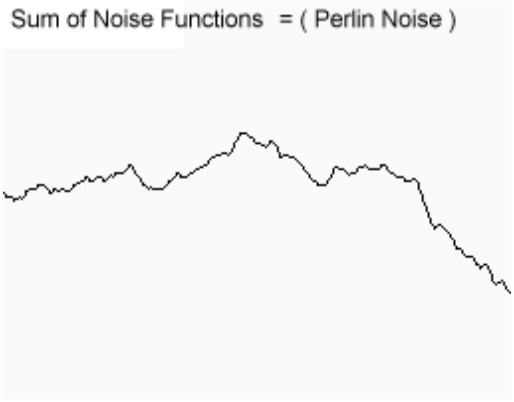


Fig. 26: “You may even imagine that it looks a little like a mountain range.”

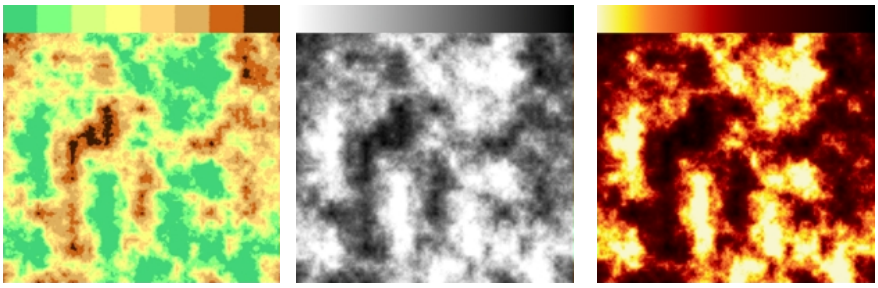


Fig. 28: Gradient mapped textures from [1]

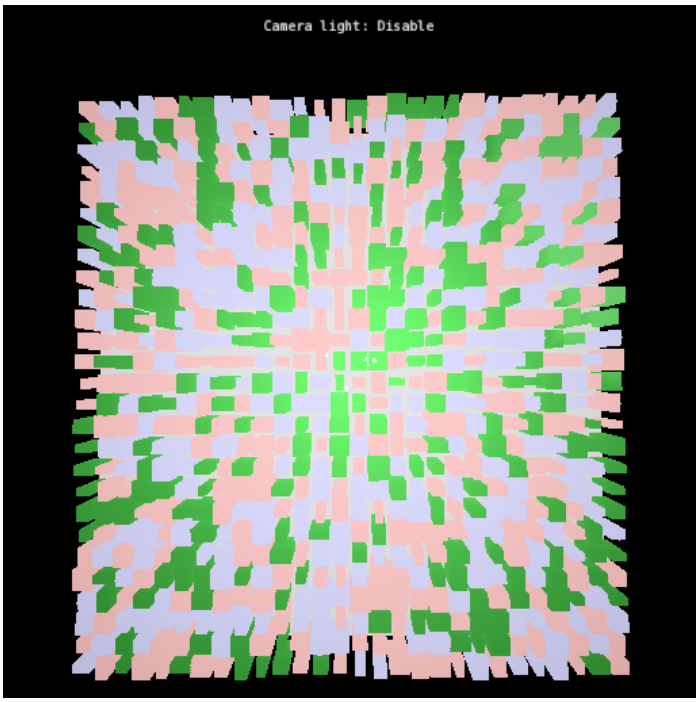


Fig. 29: Objects Placed following a noise function

type for one block, the system randomly chooses one from a set of buildings of that type.

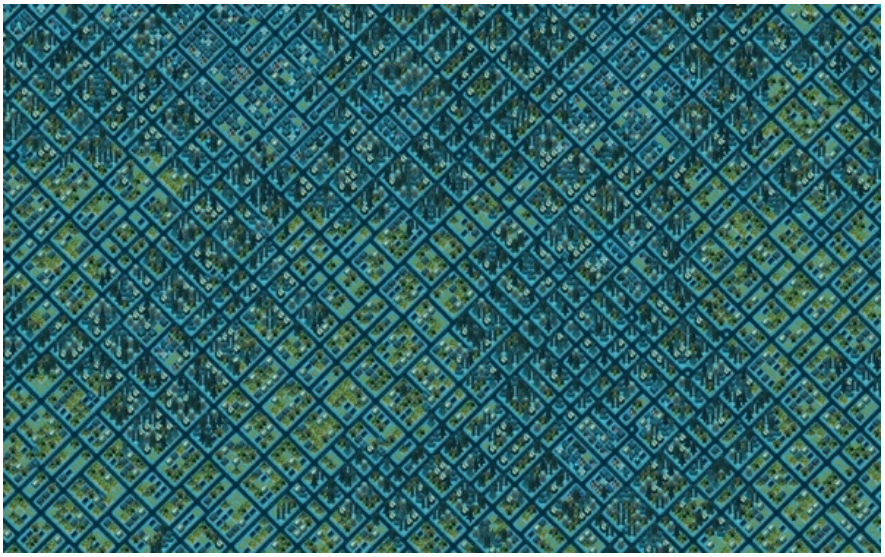


Fig. 30: Objects Placed with a noise plane from [1]

3.4 Visualization Improvement Techniques

This section will present other techniques that can be applied to achieve the performance needed for this work.

3.4.1 Level of Detail Level of Detail (LOD) 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.

This concept is easy to understand and implement if we look at the example in the Figure 31. 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.

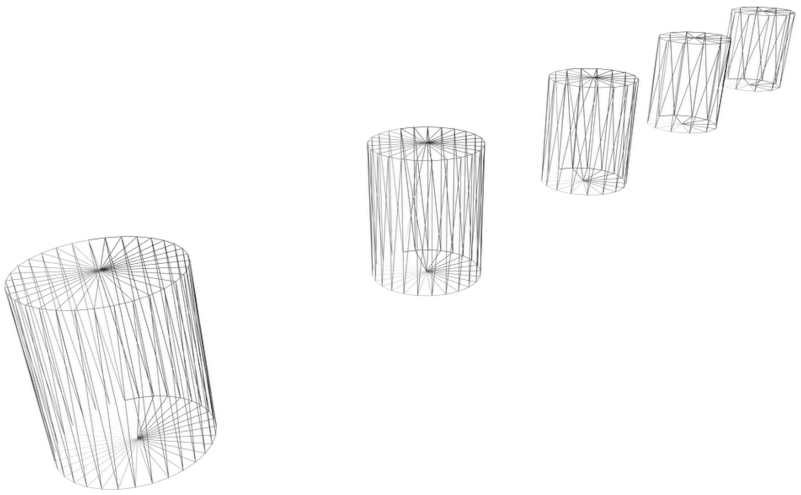


Fig. 31: LOD example

3.4.2 Occlusion Culling (OC) is another technique that is used to improve performance. It involves determining the faces that are not visible at each point, so that they can be removed from the pipeline.

This technique is usually done automatically by the GPU and applied to occluded faces behind other objects or out of the viewing frustum.

If this concept is applied before the generation of the objects, and prevents the inclusion of large amounts of geometry through the pipeline, we can make a large improvement on performance. Figure 8 is a good example. Here only the buildings that are visible from the current point are generated.

3.5 Conclusion

The systems presented (Section 3.1,Section 3.2) show ways to generate and visualize large volumes of geometry, in this cases applied to urban models. While CityEngine [11] aims to allow the users to create large and realistic urban models, where they give, in the limit, total control to the users. Undiscovered City[5] is much more a visualization tool, it generates the model automatically for the user to explore.

From these works there are some ideas to explore. The idea of immediate feedback that is implemented in CityEngine, with sliders, is a good input to my work. This helps the unexperienced users to easily see the how their code impact the results. Also the commands they have on the grammar could be an helpful inspiration for the design of my API.

From the Undiscovered City system, since they also have massive amount of geometry, how they tackle this problem is very important source of inspiration as well.

The presented procedural modeling techniques are important for the procedural generation of geometry, such as Fractals (Section 3.3.1), and Noise (Section 3.3.5), and will be explored and applied during the development of this work. Additionally, techniques related to visualization, such as level of detail (Section 3.4.1), was presented and will be explored to help improve performance.

4 Objectives

Nowadays new cities are being built completely from scratch. An example of these cities is the city of Maasdar[3] in Abu Dhabi. This city, designed by Foster and Partners, is currently being built in the desert at an estimated cost between 18 and 19 billion US$. Whoever is responsible for a project of this size can not afford to make any mistakes. To designs this projects it is necessary to create models, but in contrast to the relatively small size of a model of a single building, in this situations an entire city has to be modeled. Therefore, it requires generation and visualization of very large amounts of geometry.

The overall goal of this work is to build a GD tool that solves the performance problems that raise with the generation and visualization of large amounts of geometry. It should be an easy to use tool, with very high performance that will be focused on model visualization, avoiding features regarding interactive model manipulation.

It also should be able to support Immediate Feedback, i.e. allow the users to quickly see the results of the changes they make, for much larger models than the current GD tools can handle. This system should also provide a significant amount of geometric primitives such as *boxes*, *cylinders* or *spheres* that will allow users to model a large variety of shapes.

It will also provide a programming interface, that is how users will interact with the system. It should be simple and easy understand, yet broad and powerful to give the users freedom to create. This will be the visible part of the system together with the visualization window.

After, there is the **GPU communication** module that implement the functions provided. This module will generate the geometry description, create the windows and transfer the data to the GPU. This module will implement a set of techniques that will

The **GPU pipeline** is where the geometry will be generated and is explained in Section 5.2.

This work is being developed in the context of the Rosetta that is also a GD tool that helps architects and designers to develop their work procedurally. Rosetta is an extensible IDE based on **DrRacket** and built in Racket.

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.

On the next section will be presented related works that have similar objectives.

[3] `http://www.masdar.ae/en/masdar/detail/masdar-city-free-zone`

5 Architecture

This section present the architecture of the solution that we propose, but before will introduce some technology that is used in the solution.

5.1 Graphic Tools

There are several Application Programming Interfaces (APIs) for graphic content creation, but the most 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 the state of 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, this 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.

5.2 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 after version 3.2.

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 32. While some of steps are fix and are automatically executed, other steps are programmable, which allows the developers to program directly 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[14].

First the model is created from geometric primitives and it is the input for the pipeline (*Vertex Data* on Figure 32).

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

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

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

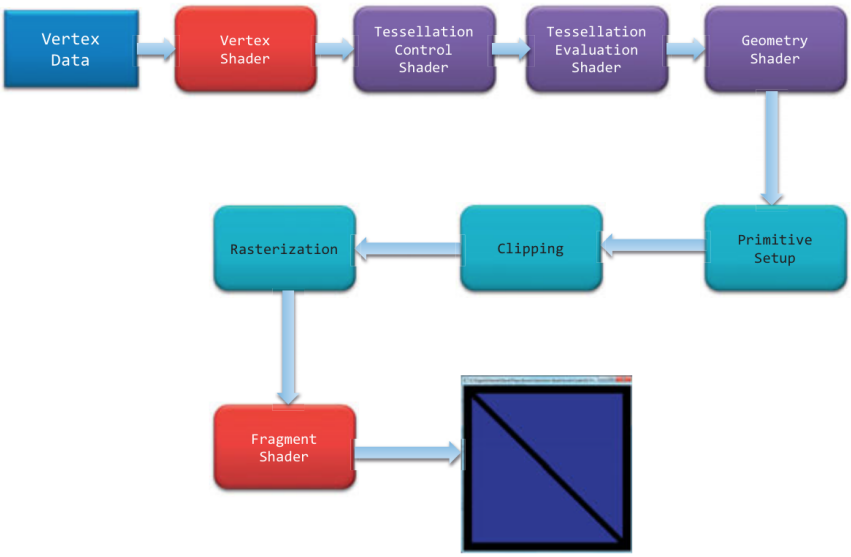


Fig. 32: OpenGL Pipeline [14]

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 still can be rejected and never update its associated pixel location" [14].

5.2.1 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 to 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 and the number of vertices after this step is the same as before.

5.2.2 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.

5.2.3 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 [14]. 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 33 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 correspond with one side of the triangle and one inner value. Each outer value defines the number of divisions that its correspondent side has.

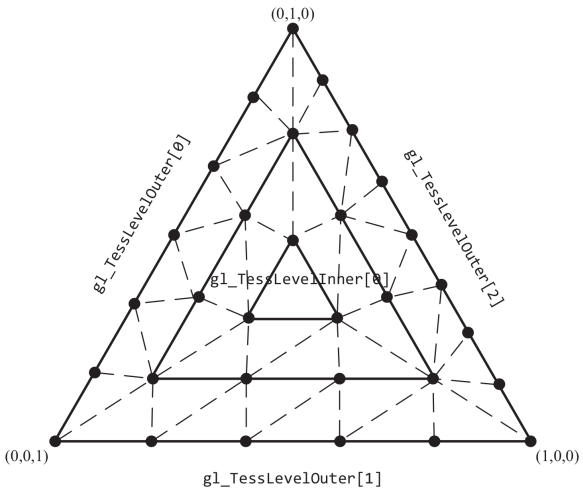


Fig. 33: Example for tesselation control with triangular domain [14]

5.2.4 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.

5.2.5 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.

5.2.6 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.

5.3 The Proposed Solution

This work will follow the architecture described in Figure 34. There will be a module in Racket that will provide a Racket interface for the rest of the system using *Racket FFI*. Is through this interface that users will interact with the system. This will be a layer that will not have an impact on performance.

The second step is the OpenGL layer, that implements the Racket interface and then creates the window and manages user input. The functions provided to Racket will create here the description of the geometry that will be *amplified* in the next phase. This description is one GL_POINT that represent each geometric primitive and is embedded with an array of floats that encode the position, the type of geometry and specific information like size or number of sides. For example, to create a cylinder, instead of generating all the points that represent a cylinder, that can have 2×32 points, each one with

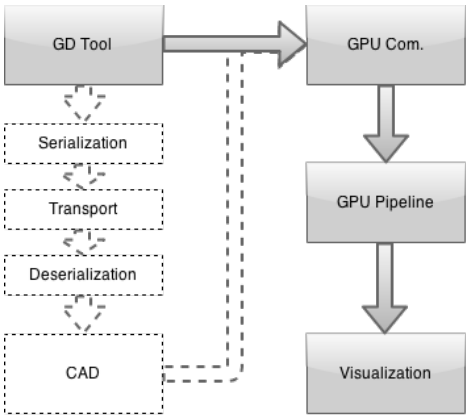


Fig. 34: High Level Architecture

three values so 193 values in total, we generate a description with only 7 values that encode the primitive type, three values for the position, and three values for the size in each direction.

In the last step are the shaders, where most of the work is done. This receives the small description of the geometry and generates the primitives to be drawn. To achieve this, it is applied the concept of geometry amplification. As explained in Section 5.2.5, this method has limitations that could make an impact on how the geometry generation is implemented. However OpenGL guarantees support for at least 256 vertices which is enough to generate the majority of geometric primitives. Since this problem is hardware dependent and GPU hardware is getting more powerful this should not be a problem in the near future. To achieve high performance with this system, will be explored and implemented within this module the concepts of level of detail (LOD) and object culling. The first is related with the detail which each object is generated in relation with the camera position, generating objects with high detail when they are close to the camera and to progressively lose detail when move away from the camera. At the same time, objects that are partly or completely covered by other objects are generated in order to decrease the detail or even prevent them from being generated.

This architecture significantly reduces the amount of data that is moved between layers and takes advantage of the power that recent GPUs have. In order to validate this architecture, one prototype has been implemented. This prototype currently supports a subset of the geometric primitives: cubes and cylinders.

For instance the following code results in the Figure 35 that is a procedural generated model of a city with 40k buildings. This example was generated with the current prototype with the following Racket code:

```
1  (define (building x y z w l h)
2    (let ([h1 (* 0.7 h)]
3          [h2 (* 0.4 h)])
4      (begin
5        (box x y h1 w l h1)
6        (cylinder x y (+ (* h1 2) h2) (* 0.7 w) h2))))
7
8  (init 1000)
9
10 (let ([grid-size 30.0])
11   (for* ([xi (in-range (- grid-size) grid-size 0.3)]
12          [yi (in-range (- grid-size) grid-size 0.3)])
13     (building xi yi 0.0 0.1 0.1 (random))))
14
15 (start)
```

The *init* command makes an initialization of the system and its argument is an estimation of the number of primitives just for optimization purposes, the *start* command creates the window and starts the visualization. In between the

model is created, in this case the code that creates a squared city model with buildings of random height.

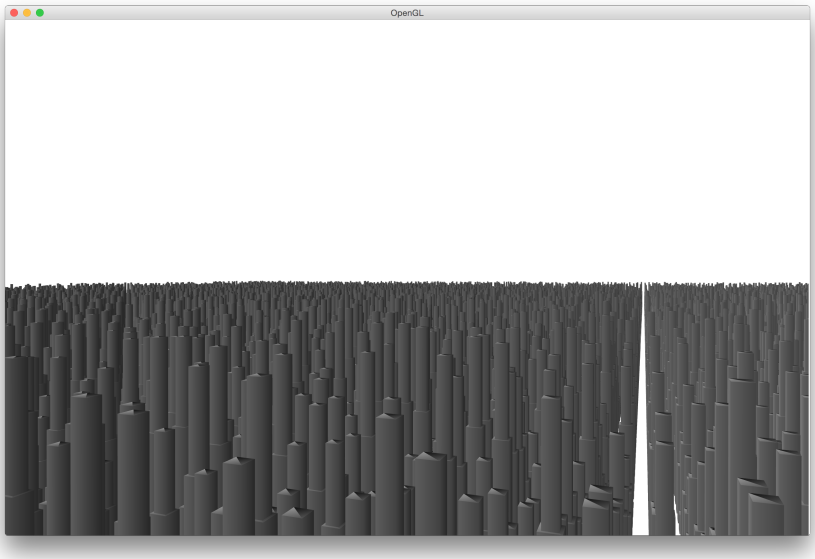


Fig. 35: City with 40k buildings

6 Evaluation

This work will be evaluated in two distinct concerns: *functionality* and *performance*.

To evaluate this system will be implemented an interface in Racket to be tested in the context of Rosetta. Rosetta is a GD tool that is largely used by a community of architects that use it to create large models. They have a large amount of examples to be run for comparison. I will use this same set of examples to benchmark my system.

This will show how broad is the system functionality, and it should implement a large set of functionality that allows the generation of a significant set of the examples. It will also test the correctness of the system, by allowing the comparison and to check if the examples generate the exactly matching results.

And performance is the main concern throughout all this work, and it will be evaluated by comparison with Rosetta's several backends with the same benchmarks.

7 Conclusions

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, more complex models. This, unfortunately, creates a performance problem because 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 designers, 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 need.

My solution shortcuts the Rosetta traditional pipeline to improve performance, doing that by using Rosetta just as an interface and transferring to the GPU as most of the processing as possible.

In order to evaluate the planed architecture of our system, one prototype was implemented that already creates boxes and cylinders without transformations that allows the creation of the city example in Figure 35.

In the future, I will explore how to implement the rest of the primitives, without losses in performance and how to introduce transformations to the objects without adding much data to the current primitive description set.

References

1. How to use perlin noise in your games. `http://devmag.org.za/2009/04/25/perlin-noise/`. Accessed: 2015-01-15.
2. Perlin noise. `http://freespace.virgin.net/hugo.elias/models/m_perlin.htm`. Accessed: 2014-10-16.
3. H Abelson. Aa disessa. *Turtle geometry*, 1982.
4. David S Ebert, Forest Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and modeling: a procedural approach*. 2002.
5. Stefan Greuter, Jeremy Parker, Nigel Stewart, and Geoff Leach. GRAPHITE 2003 Real-time Procedural Generation of ‘ Pseudo Infinite ’ Cities. 2003.
6. George Kelly. An Interactive System for Procedural City Generation. 2008.
7. George Kelly and Hugh Mccabe. A Survey of Procedural Techniques for City Generation.
8. Menezes Leit. Programação para Arquitectura. 2012.
9. Benoit B Mandelbrot, Dann E Passoja, and Alvin J Paullay. Fractal character of fracture surfaces of metals. 1984.
10. Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *ACM SIGGRAPH 2006 Papers on - SIGGRAPH ’06*, page 614, 2006.
11. Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. *Proceedings of the 28th annual conference on Computer graphics and interactive techniques - SIGGRAPH ’01*, pages 301–308, 2001.

12. Pedro Palma Ramos and António Menezes Leitão. Implementing Python for DrRacket. In Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões, editors, *3rd Symposium on Languages, Applications and Technologies*, volume 38 of *OpenAccess Series in Informatics (OASIcs)*, pages 127–141, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
13. Daniel Shiffman. *The Nature of Code*. 2012.
14. Dave Shreiner, Graham Sellers, John M Kessenich, and Bill M Licea-Kane. *OpenGL programming guide: The Official guide to learning OpenGL, version 4.3*. Addison-Wesley, 2013.
15. Eric W. Weisstein. Cellular automaton. From MathWorld–A Wolfram Web Resource.`http://mathworld.wolfram.com/CellularAutomaton.html`. Accessed: 2015-05-16.

A Appendix

A.1 Work Scheduling

	May	**Jun.**	**Jul.**	**Aug.**	**Sept.**	**Oct.**	**Nov.**	**Dec.**
State of the Art reading								
Solution Development								
Implementation								
Tests and Evaluation								
Writing								
Review								

Table 1: Work Scheduling