Procedural Generation
City Modeling

Artur Alkaim

Instituto Superior Técnico, Universidade de Lisboa
arturalkaim@tecnico.ulisboa.pt
http://tecnico.ulisboa.pt/

Abstract. Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Keywords:
Graphics, OpenGL, Generative Design, Procedural Generation

1 Introduction

As technology evolves people have more powerful devices and they want to take advantage of that having more realistic experiences with larger, more detailed and complex contents. And this is observable in the graphic contents. With the recent extra high definition on screens and the computational power of the machines beating records, the graphic content has to follow up that characteristics in quantity as well as in quality. The issue is that the manual content generation takes a long work time from architects and designers to achieve this quality, thus implying high costs.

Graphic contents are mainly used for entertainment, both in the gaming and movie industries, but it is also used in a lot more 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. Also in this field they face also the problems that raises from the modeling of really big sets of objects and forms manually. *This work focus on this problem of content creation for the fields of architecture and design.*

The obvious answer to this problem of manual content creation costs is to contract more architects or designers to each project to increase the production, but experience have shown that this solution is not scalable, that means that double the number of architects or designers working in a project will not double their overall productivity. Also this solution has a big impact on costs, that would take immediately out of the market new producers with less resources.

A solution for this problem is the use of generative design. That is a design method that is based on a programming approach which allows architects and designers to model complex shapes with significantly less effort.

Although most computer-aided design (CAD) applications provide programming languages for generative design, programs written in these languages have very limited portability. This languages, such as AutoLisp, C++ or Visual Basic, are not pedagogical and are difficult to use even to experienced programmers, problems that create barriers to adherence to this approach by users specially to those that normally are not used to code.[14]

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

One big difference from traditional approaches is that the users do not see the result of their program while they code. They should follow a code-execute-visualize loop where they make changes in the code, execute the code and visualize the resulting model. They also want to 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 see the results of the changes they make in the resulting model. 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.

With this rises the problem of performance. The architects and designers implement their models on GD tools that generate the geometry. Then, as Figure 1 shows, the all geometry data is serialized and the data is transfered through sockets. This data have to be deserialized on the other side within the CAD application. This CAD application takes the objects and processes them adding their specific information to the geometry. At last the geometry is moved to the GPU that renders the model. All this steps are time-consuming mostly caused by the large amount of data that is repeatedly transfered. There is also another problem, CAD applications are built for manual modeling mainly, and are not prepared to receive large amounts of geometry very fast. Running the code is much faster than manual

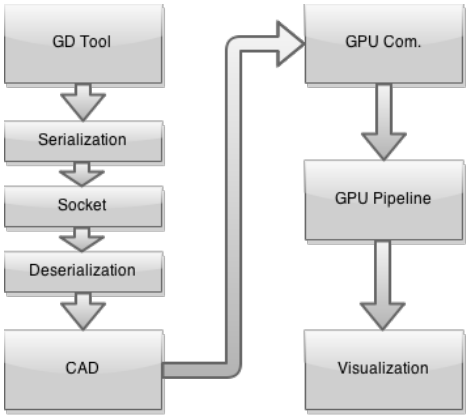


Fig. 1: Common Generative Design Pipeline

modeling, so the user is able to create massive amounts of geometry fast that 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. Users sometimes have to wait a lot of time for the model to be rendered.

This work proposes a solution to this problem. It does so by jumping over some steps while drastically decrease 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 and with that eliminate the communication connected with it. Another action is to reduce the amount of data that is transferred and it is achieved by transferring only a very concise description of the geometry that is generated by code running on the GPU. To implement the generation of the geometry procedural techniques will be studied and applied.

2 Overview

This Section will provide an overview over this topic providing background. The Section 3 will address the objectives for this thesis work. Section 4 will explore different yet related work to this program. Section 5 will describe the architecture of the proposed solution. Section 6 will explain how this solution will be evaluated and I will conclude on Section 7.

2.1 Procedural Generation

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 graphics creation and sound (music and synthetic speech).

The key property of procedural generation is that in describes the data entity, be it geometry, texture or effect, in terms of a sequence of generation instructions rather than as a static block of data [9]. This allows the production of big volumes of detailed, and high quality graphic content without the costs, both in time and price, of manual content creation.

In the following sections some techniques of procedural modeling will be detailed.

2.2 Fractals

A fractal is defined in [6] 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. From trees, mountains, coastlines to 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 [11]. In [6] is proposed to think of

fractals as a new form of symmetry, *Dilation Symmetry*, which is when an object is invariant over a change of scale. This invariance migth 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. As clouds, tree branches and some vegetables as shown in Figure 3. These examples are fractals.

Fig. 3: Fractals in Nature

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

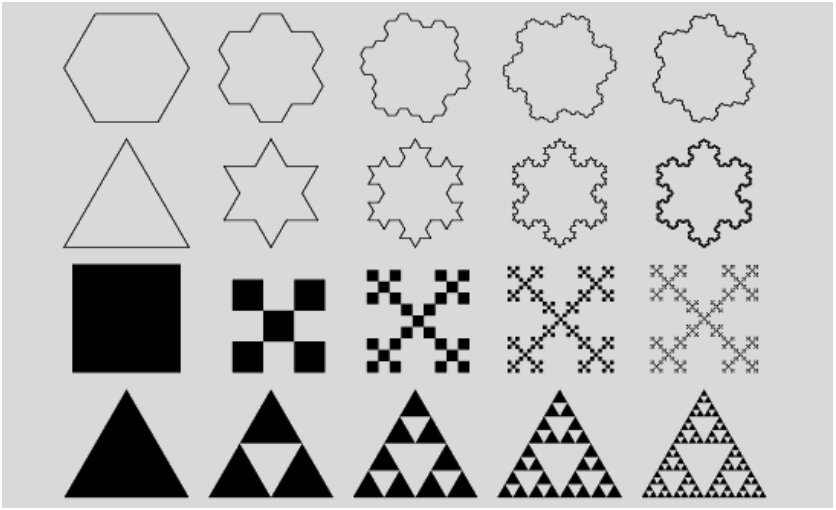


Fig. 4: Geometric Fractals

In the Figure 4 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's clear that the detail that is presented in each iteration increases as the scale changes. To try to mesure this evolution there is the concept of $FractalDimension$ in which the detail in a pattern changes in comparison with the scale in which it is measured.

As stated before, the world is visualy very complex, so when synthetizing worlds, "*complexity* equals *work*"[6]. This work can be done by the programer/artist or by a computer. Some cases are not easy to model procedurally. This is not the case of the fractals, as being defined as a simple mathematical function, it's relatively easy to implement a procedure to model fractals.

This techique is used to model many natural forms that present fractal properties. Mountains, as an instance, are usually modeled using of fractals. Other natural forms that present fractal properties are trees, river systems, ligthning or vascular systems in living beings.

2.3 Cellular Automaton

It's a model of a system of cells within a grid with a determined 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 with the state of the neighbouring cells. The neighbourhood of the cell can be defined in many different ways, the most common is the use of the adjacent cells.

In 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 reffered by their *Wolfram code*, defined by Wolfram.

A common initial state for yhis elementary cellular automata is a random line. But to able to compare the results between rules and get clean results other option is to start with a line with zeros except the middle cell with one. Applying this second option and the following set of rules (the rule 30):

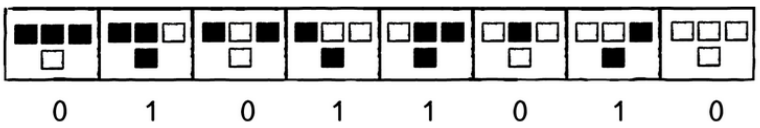


Fig. 5: Example Production Rules[15]

we get the pattern in the Figure 6 that represents the evolution of this Cellular automaton over generations:

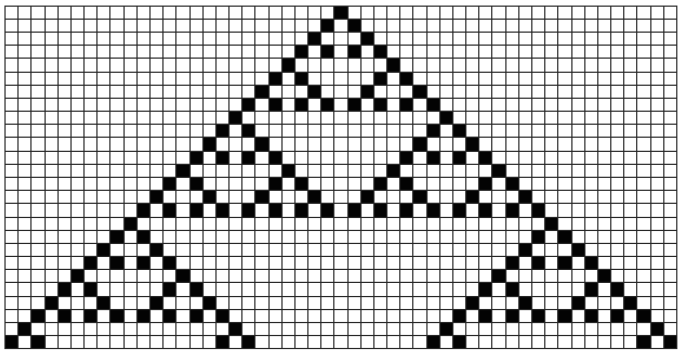
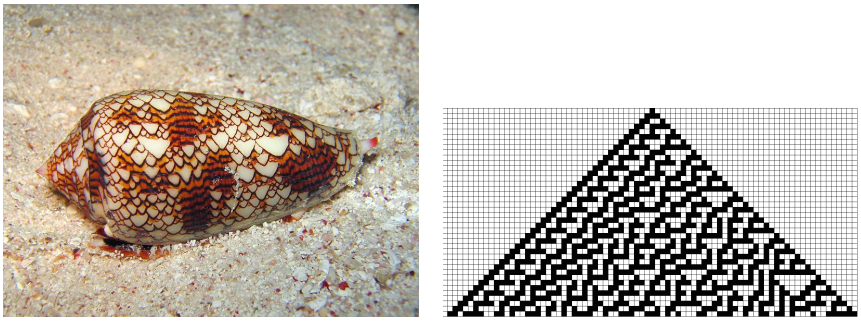


Fig. 6: Sierpiński Triangle, rule 90

In Figure 6 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 occurs 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 7a shows a natural texture on a Textile Cone Snail that looks like the patterns formed with the cellular automaton in the Figure 7b.



a) b)

Fig. 8: Example of the representation of natural patterns with cellular automata. a) Natural Shell, image from [15]. b) Pattern formed with the rule 30.

2.4 L-Systems

Lindenmayer Systems (L-Systems) are a class of string rewriting mechanisms, originally developed by Lindenmayer as a mathematical theory for plant development.

One L-System is a type of a formal grammar and a string rewriting system that is capable of describe the behaviour of plant cells and model the growth processes of plant development.

It 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 is't applied in this seed the set of production rules, that change the initial string and 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 wish grows the size of the set even larger.

This L-Systems are used to produce natural growth of vegetation (Figure 9), and the generation of Fractals.

Fig. 9: 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 $F + F - -F + F$. In 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 [4], with the symbols being interpreted as moving instructions to the turtle, is a simple way to understand. If "F" means forward and the symbols "+" and "-" are interpreted as rotations counter-clockwise and clockwise respectively by a predefined angle.

Using the given example, and setting the angle for the rotation to 60° the result is Figure 10.

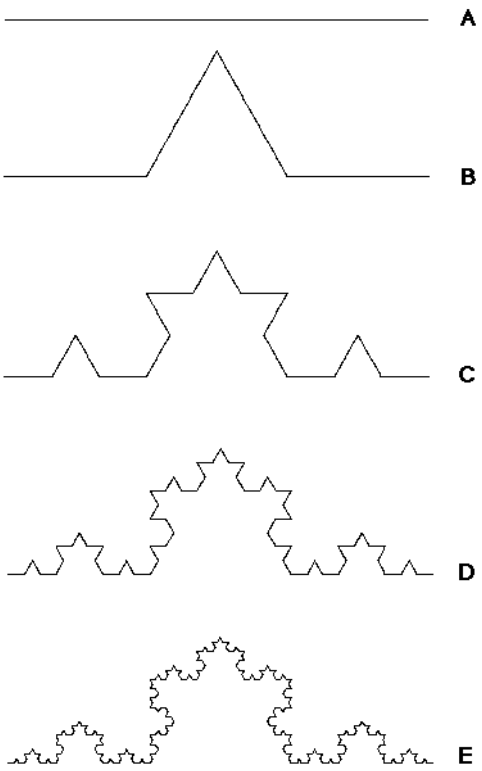


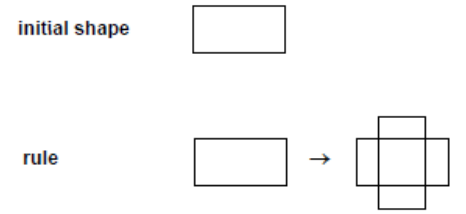
Fig. 10

2.5 Shape Grammars

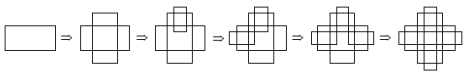
Shape Grammars can be considered grammars for design. In stead of having symbols or letters as components of the alphabet, it has shapes that can be in 2D or 3D. And have production rules that are composed by this shapes, and specify the evolution of the system. With this process, similar to the L-Systems 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 that are previously defined.

The Figure 12 exemplify one shape grammar, with one rule and the evolution of the application of this rule to the shapes iteratively. In this image, it's shown that from very simple initial shape, can be generated a complex from with a few iterations.



a)



b)

Fig. 12: a) Grammar Tiles b) Recursion steps

This is applied to the generation of buildings in the CityEngine 4.1 system, using 3D blocks for the main form, and 2D shapes to design the facades.

The Figure 13 shows a simple building that I modelled using CityEngine and it's CGA Shape Grammar. But CGA is powerful enough to model much more complex buildings like the Figure 14.

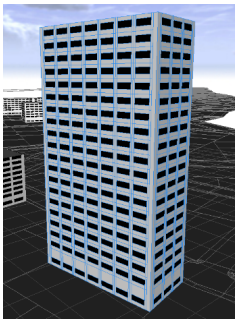


Fig. 13: Simple Building

Fig. 14: Complex Building [12]

2.6 Tiling

A common solution to give realism to 3D objects is the application of textures. One problem is that this textures takes a lot of memory space. To work around this problem an easy solution is to have a small texture piece, a tile, and repeat throughout the objects. And this is called tiling. Or as defined in [17]: "A plane-filling arrangement of plane figures or its generalisation to higher dimensions.". This means, the result of constructing a plane from a finite set of "tiles".

If this technique is use naively commonly results in not very homogeneous textures, it depends much on the set of tiles that are used. If the borders of all the tiles are all the same the result is always homogeneous. But if the borders are very different the chances to have a not uniform texture rises. The Figure 15, from [3] shows how a bad structured tiling system produces a not homogeneous texture.

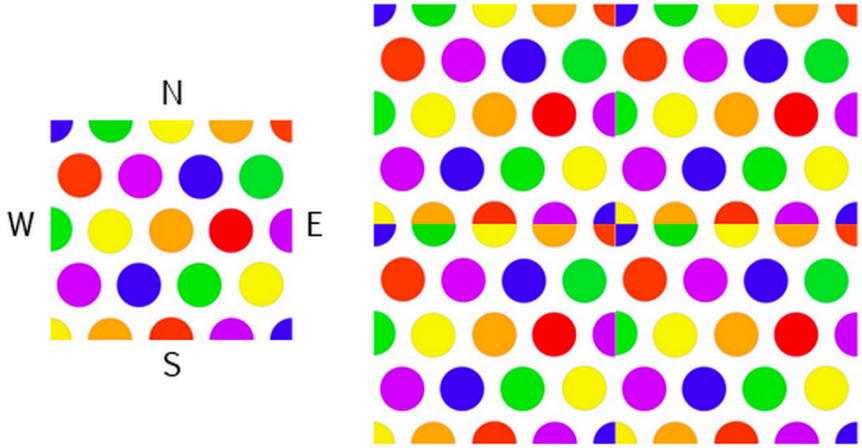
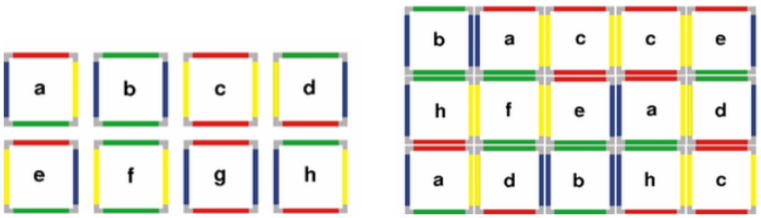


Fig. 15: One tile and an irregular pattern

To make uniform planes, the boundary of each tile must be coherent, i. e., the borders of connecting tiles have to match. Given a single tile, the so-called first corona is the set of all tiles that have a common boundary point with the tile (including the original tile itself). From that, the simple method to create a homogeneous texture is to connect each tile with one that belongs to it's first corona.

The easy, most simple solution is to make sure that all the tiles have the same borders all around. With this property it's guaranteed that any created texture will be homogeneous. But this solution doesn't provide much irregularity and the results present repeating patterns.

Wang Tiles [5]. It is a solution named after Mr Hao Wang that predicted that tiling was not possible. It received his name, not only for him being wrong, but because the prove that this is possible uses much of the work he did trying to prove the impossibility. This process allows tiling with an arbitrary number of different vertical and horizontal borders and from that calculate the set of tiles that are needed to create a full texture without inconsistencies.



a) b)

Fig. 17: a) Eight Wang Tiles b) Portion of a plane

So the process assign colors to the tiles borders and then matching colored borders are aligned forming a plane.

As you might have noticed, the inner content of the tiles are not a problem. As we are trying to create the uniform textures by arranging this smaller pieces only the borders matter, so we can create a set of tiles with the same borders and whatever inner content we want. With this technique the result can be much more irregular and more natural looking textures.

Corner Tiles. [10] The vanilla wang tiles have problems in the diagonals that are not taken into account. They are the confrontation between four tiles which leads to less homogeneous texture if we the borders don't match. By using the corners, the problem goes to the sides, that despite being larger, are only the confrontation between two tiles and therefore it leads to a more homogeneous texture.

2.7 Noise

"To generate irregular procedural textures, we need an irregular primitive function, usually called noise" [6]. It's a pseudorandom function that gave the goal to break 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 1982 with the aiming 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's generated and we would like that the it stays the same, frame after frame. This is achieved with the use of inputs for this functions that with the same input returns always the same output sequence.

The properties of an ideal *noise* functions are as follows [6]:

- *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* doesn't 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, it's generated a sequence of values that are interpolated to generate 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. This layers are called *Octaves* and the ratio between amplitude and frequency of the layers can be expressed as a constant known as *persistence* [8]. With the result we can create a texture that looks natural and with fractal like structure.

For instance, the Figure 18 shows the result of the interpolation over six noise functions with different frequencies and different amplitudes. And the sum of all this functions is the exibithed in the Figure 19 [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 after are interpolated as a plane. This results in noisy images that are often used to model clouds, smoke and other textures with similar visual properties, Figure 21. Another application for this technique is the generation of height maps like the one in figure .

Another application for Noise planes are with *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 22. Figure 23 shows a city which the buildings where placed with the use of a noise plane. In this case the noise domain was splitted in three intervals, each one corresponds to

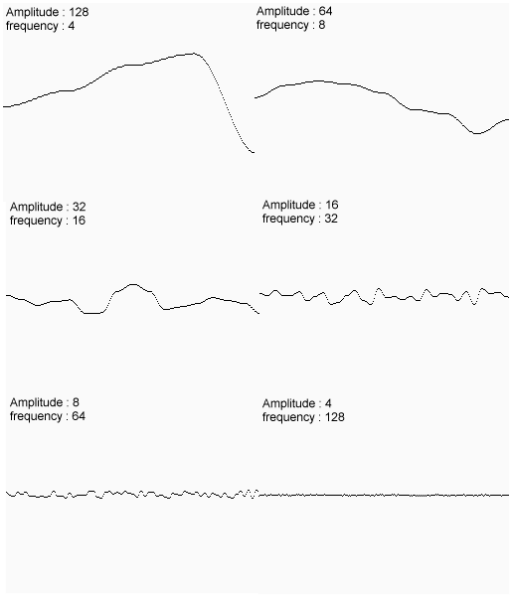


Fig. 18: Different noise functions

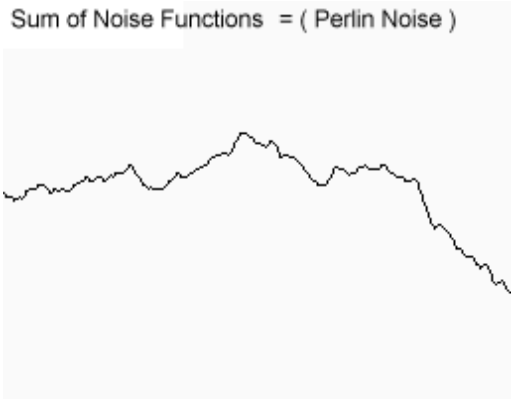


Fig. 19: “You may even imagine that it looks a little like a mountain range.”

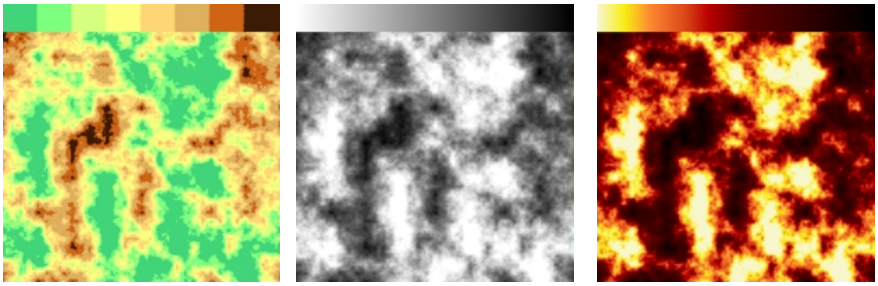


Fig. 21: Gradient mapped textures from [1]

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.

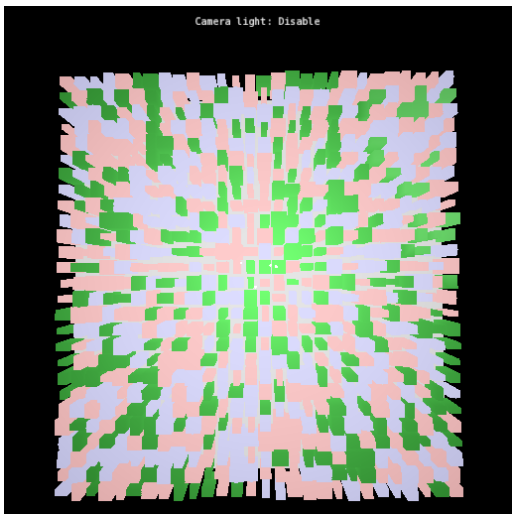


Fig. 22: Objects Placed following a noise function

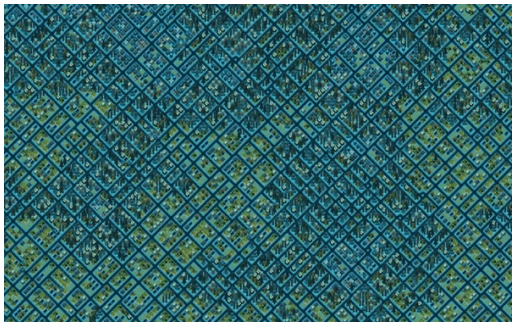


Fig. 23: Objects Placed with a noise plane from [1]

2.8 Modern OpenGL

Majors changes has been imposed to this library from it's early versions and this section covers the modern version of OpenGL, after version 3.2.

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

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

Shaders "are special functions that the graphics hardware executes. The best way to think of shaders is as little programs that are specifically compiled for your graphics processing unit". [16]

There are some steps that are performed to render an image. First the model is created from geometric primitives and this data is the input for the pipeline(Vertex Data) in Figure 24.

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

After this, 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.

Until now all steps work with vertices, and 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 of the "screen" and the generation of fragments respectively.

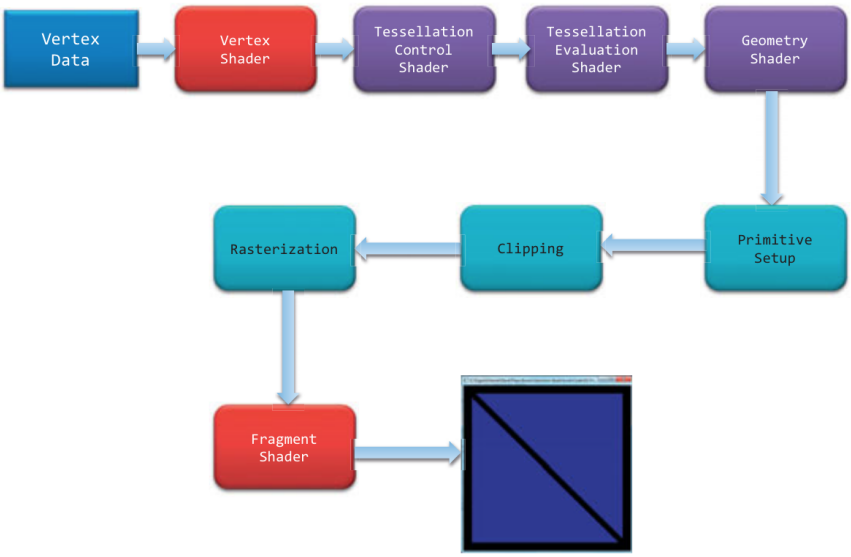


Fig. 24: OpenGL Pipeline [16]

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" [16].

2.9 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.

In this shaders are used to performed 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.

2.10 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. This is because each patch can have an arbitrary number of vertices that have to be specified before drawing opposing to the other primitives that have a specific/pre-assigned/fixed number of vertices.

2.10.1 Tessellation Control Shader This 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 [16]. To control the amount of tessellation two sets of values are set, the outer-tessellation values and the inner-tessellation values. This values define how the perimeter or the interior of the domain are subdivided respectively.

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

2.11 Geometry Shaders

Geometry Shaders are the first shaders that access the complete primitive as a list of vertices and with that it's 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.

2.12 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.

3 Objectives

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

It should be able to support Immediate Feedback 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.

The goal is to build a programming interface, that is how users will interact with the system, and it should be easy and simple to use yet broad and powerful to give them freedom to create. This will be the visible part of the system.

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 2.8.

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.

4 Related Work (17pgs)

As an active research field inside Computer Science, there is a lot of work being done in this area. In this section I give an overview of the related work that has

been carried out on procedural generation of large amounts of geometry. Since the target audience of this work are architects and designers, most of the work that I present have the same target.

The main goal of this systems is to generate models that represent entire cities, that are large by definition. With this, they face some of the same problems I face so it is important to learn about their solutions.

4.1 CityEngine [13] [12]

It's a three-dimensional (3D) modeling software developed by Procedural Inc. (now part of the Esri R&D Center). It's specialized in the generation of 3D urban environments. With the procedural modeling approach, CityEngine enables an efficient creation of detailed and large-scale 3D city models with a lot of control from the user.

4.1.1 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, bat it can be set by the user to a different one.

To implement this growth process, it's used an L-System, that computes the road network.

The Figure 25 shows the evolution of this process in a map of Manhattan. The first two on the top shows the process in different phases during the process, the middle line is the result of the process and the bottom line is the real map of Manhattan for comparison.

Fig. 25: Road Map growth

4.1.2 Buildings To implement the generation of buildings, they created the CGA Shape.

CGA Shape is a Shape Grammar that was introduced by Pascal Muller, Peter Wonka and others, in a paper called "Procedural Modeling of Buildings"[13]. 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 like Grasshopper 3D, that is nice for simple works but it's impossible to work with a slightly more complex work.

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 26.

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's possible to create complexity to the level that is desired, being able to produce high complex buildings like the one in the following picture.

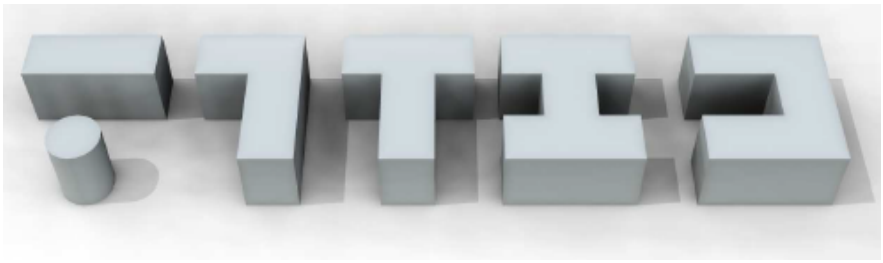


Fig. 26: Basic shapes

4.1.3 Cities The result can be an city like Figure 27, with approximately 26000 buildings.

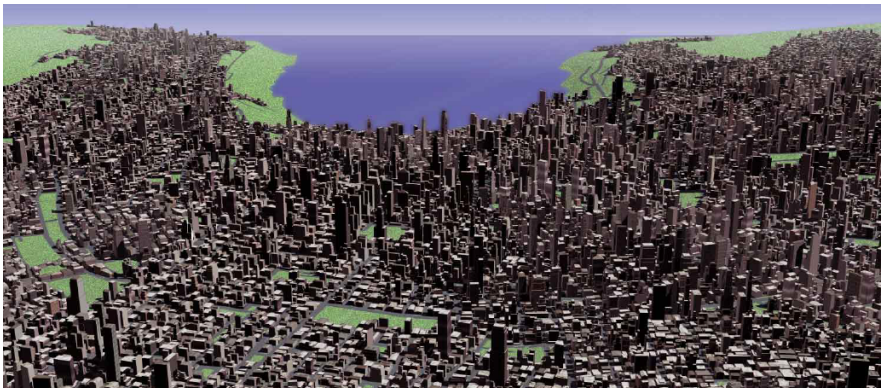


Fig. 27: City with approximately 26000 buildings.

City Engine results can be imported by Maya, to achieve better results. Like the Figure 28, that represents a 'virtual' Manhattan.

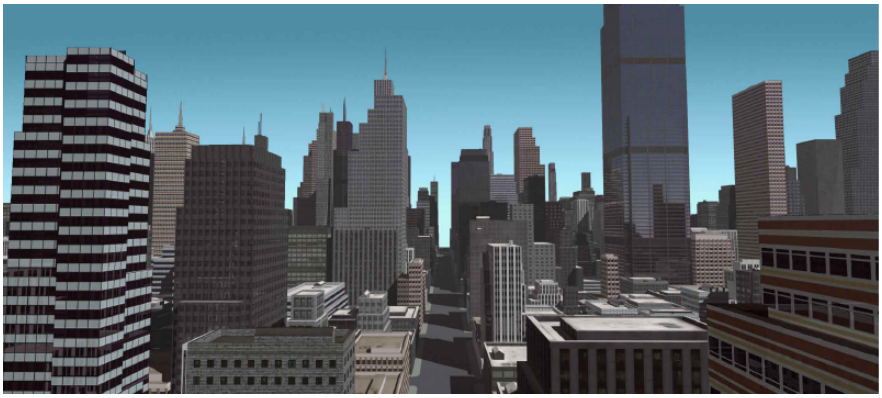


Fig. 28: City rendered with Maya.

4.2 Undiscovered City

In [7] 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 29 only the part of city that is inside the viewing range is generate.

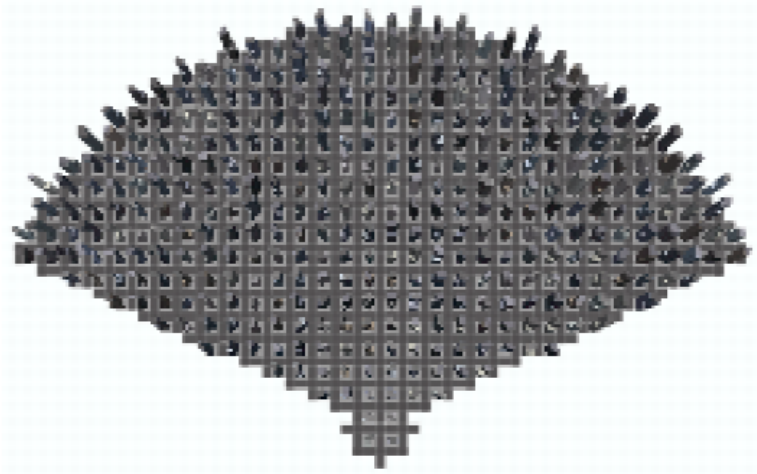


Fig. 29: Viewing Range

4.2.1 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.

After that 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.

4.2.2 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 is first is generated a floor plan. To do so, it's 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 30. 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.

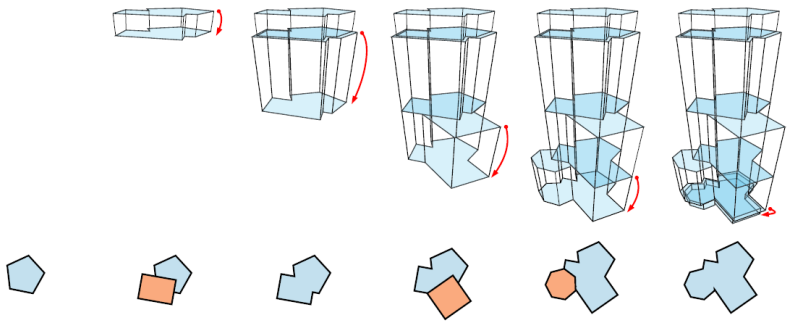


Fig. 30: buildings

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.

4.3 Comparison

5 Architecture

This work will follow the architecture described in Figure 31. There will be a module in Racket that will provide a Racket interface for the rest of the system using *Racket FFI* and is through this that the users will mostly 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 implement the Racket interface and than create the window and manage 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 per geometric primitive that represent the position of that primitive and it is also embedded with an array of floats that encode the type of geometry and specific information like size or number of sides.

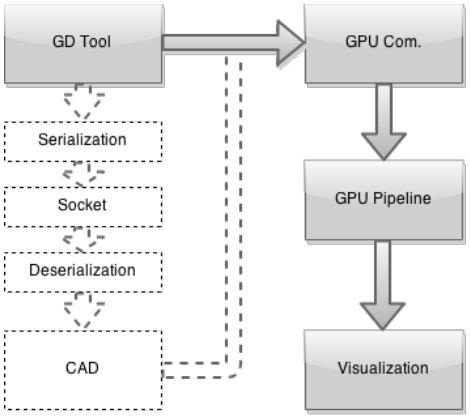


Fig. 31: High Level Architecture

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 told in Section 2.11, this method has limitations that could make an impact on how the geometry generation is implemented. However OpenGL guarantee 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 are getting more powerful I believe that this will not be a problem in the near future.

This architecture significantly reduces the amount of data that is moved between layers and takes advantage of the power that recent GPUs have.

For instance the following code results in the Figure 32 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:

```
(init 0)
(city 100)
(start)
```

The *init* command makes an initialization of the system, the *start* command starts creates the window and starts the visualization. In between the model is created, in this case with the function city.

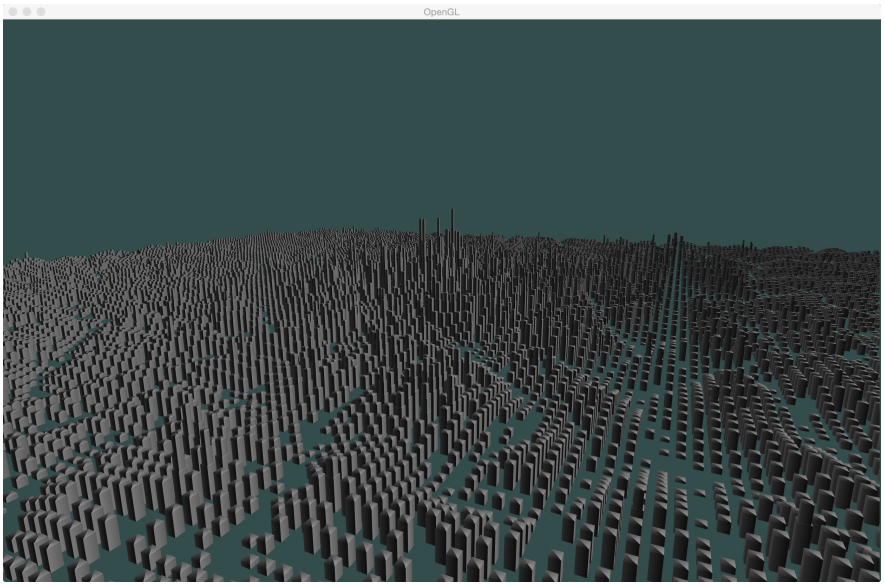


Fig. 32: City Example

6 Evaluation

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

Performance is the main concern throughout all this work so this point will be evaluated through benchmarks and comparing the time that this system takes to generate and render a set of examples to the normal pipeline.

As this work will have to provide an API, it is important to know if it is easy to use and have the needed functionality for the users to produce their models.

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.

7 Conclusions

Architects and designers increasingly use programming as a tool which helps them to develop their work. This powerful tool enables them to create faster and with greater creative freedom. With the development of their programming capabilities they begin to create larger, more complex models and to test the technology at maximum and thus realize their limitations.

The problem of performance arises because the systems being used were not built with this problem in mind. These were developed for a manual, slow usage. It was not thought at the time that a user could generate massive amounts of

geometry in seconds. As a result users have to wait for large periods of time since they run their programs until they can see a result.

Therefore there is a need to solve this problem, therefore this work is also needed given that proposes a solution that puts limits on another level allowing users to create.

This solution must have a good performance and support the most used functions that the users need to become a valid alternative to the currently used tools.

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

One prototype is implemented that already creates boxes and cylinders without transformations that allows the creation of the city example in Figure 32.

In the future, I will explore how to implement the rest of the primitives without losses in performance and how to introduce rotation 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. Procedural world blogspot. `http://procworld.blogspot.pt/`. Accessed: 2014-10-16.
4. H Abelson. Aa disessa. *Turtle geometry*, 1982.
5. MF Cohen, J Shade, S Hiller, and O Deussen. *Wang tiles for image and texture generation.* 2003.
6. David S Ebert, Forest Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and modeling: a procedural approach.* 2002.
7. Stefan Greuter, Jeremy Parker, Nigel Stewart, and Geoff Leach. GRAPHITE 2003 Real-time Procedural Generation of ‘ Pseudo Infinite ’ Cities. 2003.
8. George Kelly. An Interactive System for Procedural City Generation. 2008.
9. George Kelly and Hugh Mccabe. A Survey of Procedural Techniques for City Generation.
10. Ares Lagae and Philip Dutré. An alternative for Wang tiles: Colored edges versus colored corners. *ACM Transactions on Graphics*, 25(4):1442–1459, October 2006.
11. Benoit B Mandelbrot, Dann E Passoja, and Alvin J Paullay. Fractal character of fracture surfaces of metals. 1984.
12. 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.
13. 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.
14. 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.
15. Daniel Shiffman. *The Nature of Code*. 2012.
16. 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.
17. Eric W. Weisstein. Tiling. From MathWorld–A Wolfram Web Resource.`http://mathworld.wolfram.com/Tiling.html`. Accessed: 2014-10-16.