

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 and people get new and more powerful devices, they want to take advantage of that and have more realistic(**powerful ?**) 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 have 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 technique to experiment and model new designs, from small objects like a plate to buildings or even entire cities. So also in this field they face also the problems that raises from the modelling 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 have 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. These languages are not pedagogical and are difficult to use even to experienced programmers, problems that create barriers to adherence to this approach by users that normally are not used to code.[13]

There are several tools that aim to break down some of these barriers and facilitate the approximation of these individuals to programming.

With this rises the problem of performance because running this code is much faster than manual modeling, so the user is able to create massive amounts of geometry fast with tools like Rosetta and they usually have to wait a lot of time for all that geometry to be generated and then rendered for visualization. It takes a lot of time to generate this geometry but the communication and data transfer also have a significant role on this wait. This problem is more important in the development phase when the model is always changing and the user wants to see the effect of small changes immediately, if with each small change they have to wait a significant amount of time it has a significant impact for the work of the designers.

This work is being developed in the context of Rosetta and proposes a solution to this problem. It does it by eliminating layers while trying to transfer the minimum possible data between layers. 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. 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. Also some techniques will be studied and applied to improve performance.

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 it 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 [8]. 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.

2.1.1 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 [10]. 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 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. As clouds, tree branches and some vegetables as shown in Figure 2. These examples are fractals.



Fig. 2: 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.

In the Figure 3 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 measure this evolution there is the concept of *Fractal Dimension*

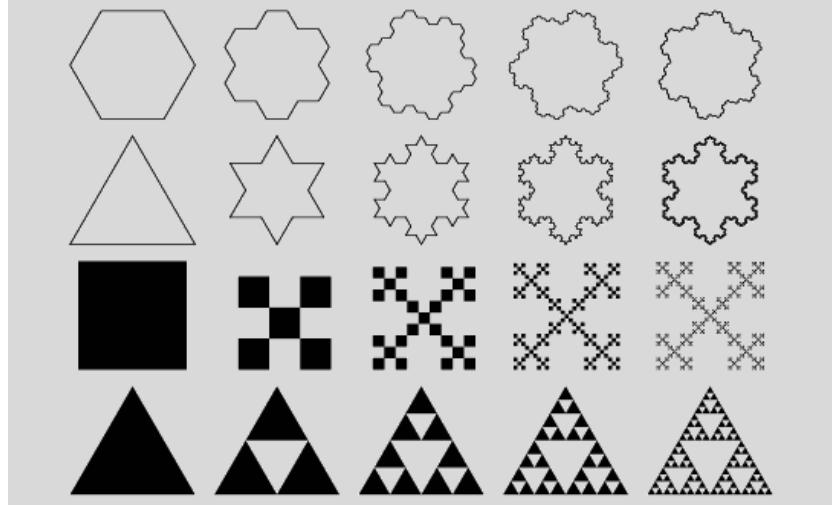


Fig. 3: Geometric Fractals

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 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, lighnting or vascular systems in living beings.

2.1.2 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 refferred 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):

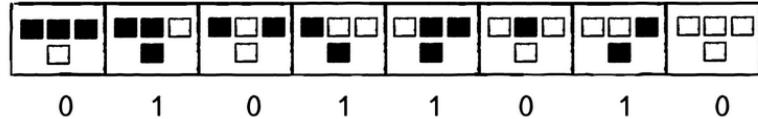


Fig. 4: Example Production Rules[14]

we get the pattern in the Figure 5 that represents the evolution of this Cellular automaton over generations:

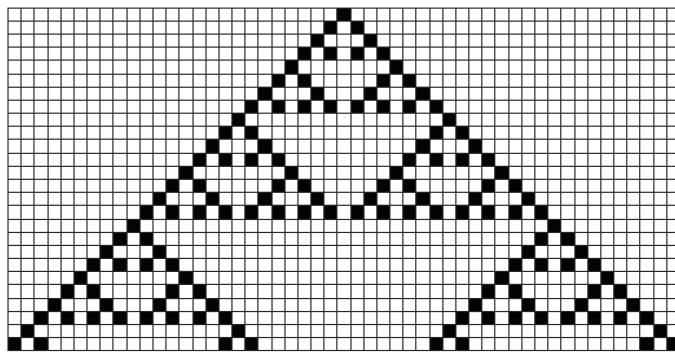


Fig. 5: Sierpiński Triangle, rule 90

In Figure 5 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 6a shows a natural texture on a Textile Cone Snail that looks like the patterns formed with the cellular automaton in the Figure 6b.

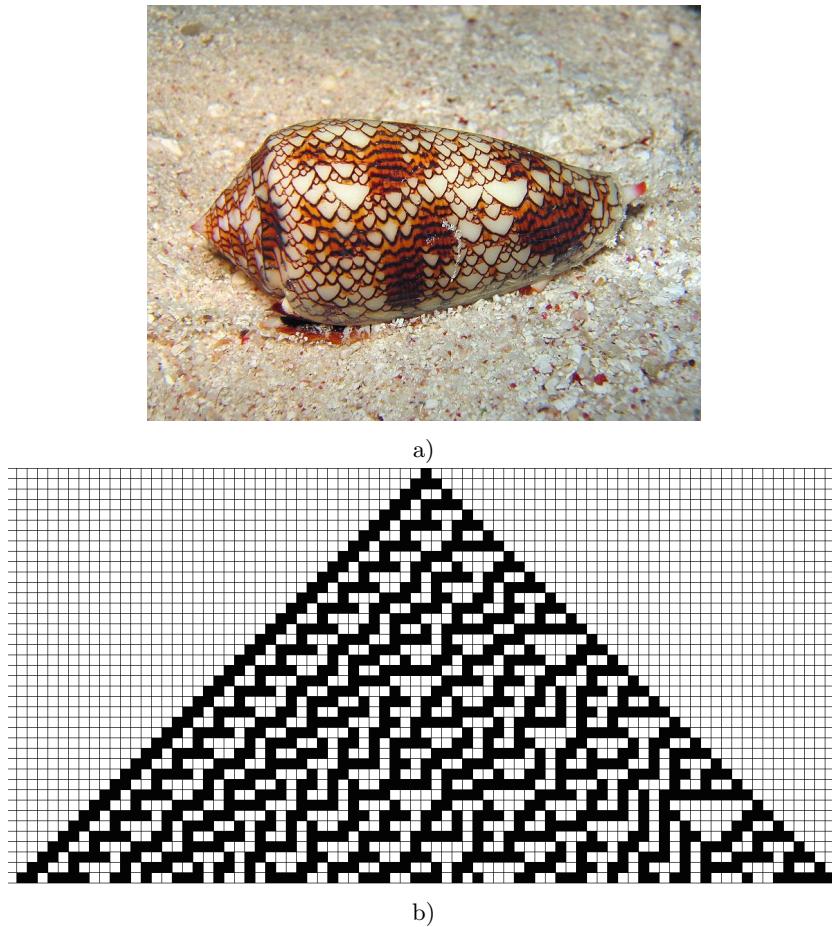


Fig. 7: Example of the representation of natural patterns with cellular automata. a) Natural Shell, image from [14]. b) Pattern formed with the rule 30.

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

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 it's 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 which grows the size of the set even larger.

This L-Systems are used to produce natural growth of vegetation (Figure 8), and the generation of Fractals.



Fig. 8: 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:

$$F \tag{1}$$

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

$$F+F--F+F + F+F--F+F - - F+F--F+F + F+F--F+F \quad (3)$$

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 9.

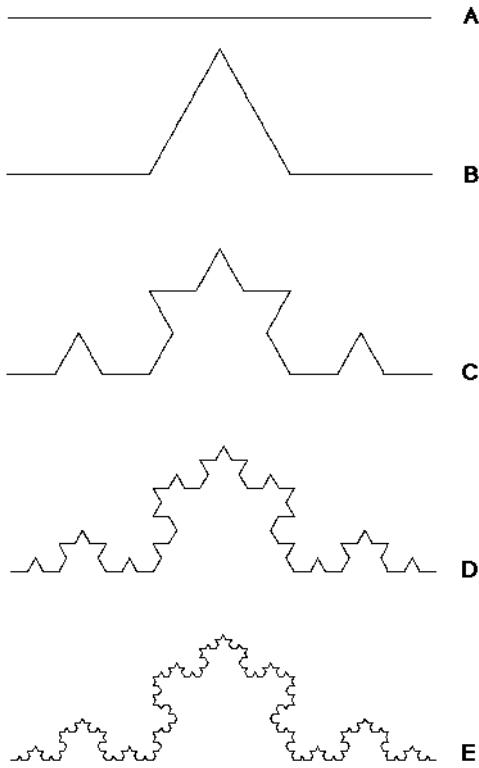


Fig. 9

2.1.4 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 11 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.

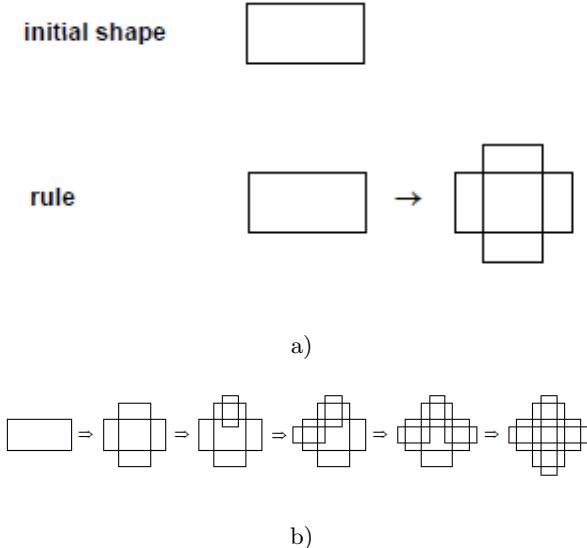


Fig. 11: a) Grammar Tiles b) Recursion steps

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

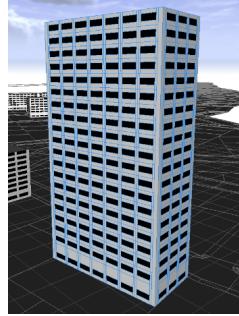


Fig. 12: Simple Building



Fig. 13: Complex Building [11]

2.1.5 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 [16]: “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 used 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 14, from [3] shows how a bad structured tiling system produces a not homogeneous texture.

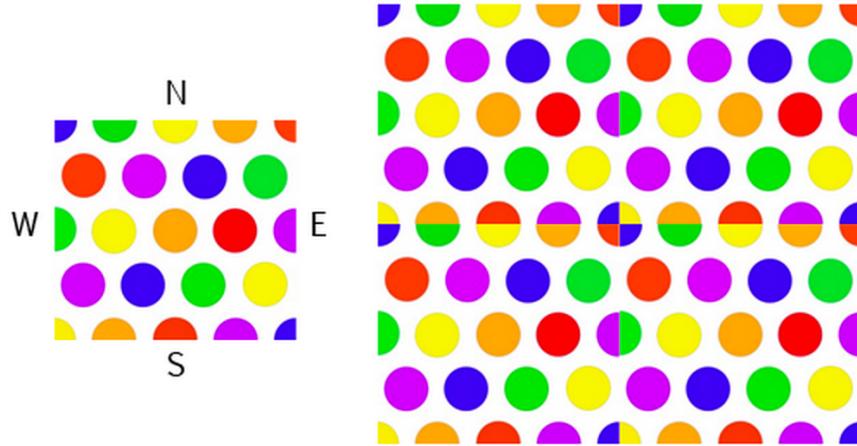


Fig. 14: One tile and an irregular pattern

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 its 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 proof 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.

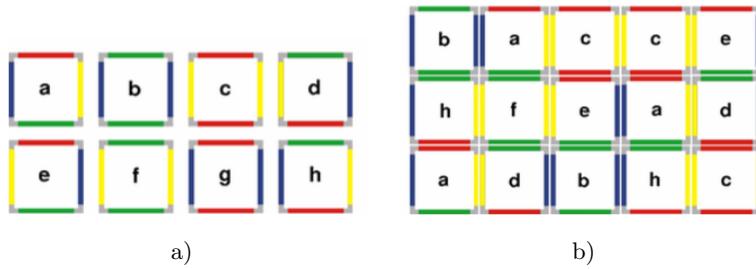


Fig. 16: a) Eight Wang Tiles b) Portion of a plane

So the process assigns 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 these 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. [9] 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 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.

2.1.6 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 pseudorandomm 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 pseudo-random 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* [7]. With the result we can create a texture that looks natural and with fractal like structure.

For instance, the Figure 17 shows the result of the interpolation over six noise functions with different frequencies and different amplitudes. And the sum of all this functions is the exhibithed in the Figure 18 [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 20. 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 21. Figure 22 shows a city which the buildings where placed with the use of a noise plane. In this case the noise domain was spilted in three intervals, each one corresponds to one type of building (commercial, industrial or residential). After setting the type

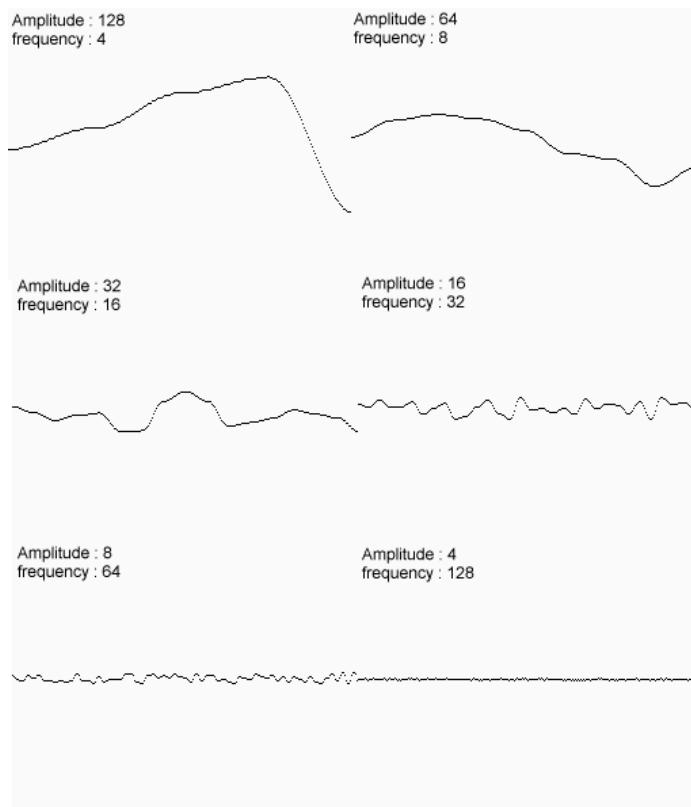


Fig. 17: Different noise functions

Sum of Noise Functions = (Perlin Noise)



Fig. 18: "You may even imagine that it looks a little like a mountain range."

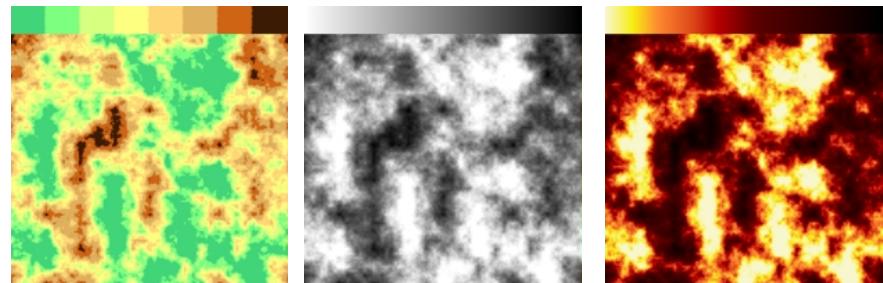


Fig. 20: Gradient mapped textures from [1]

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

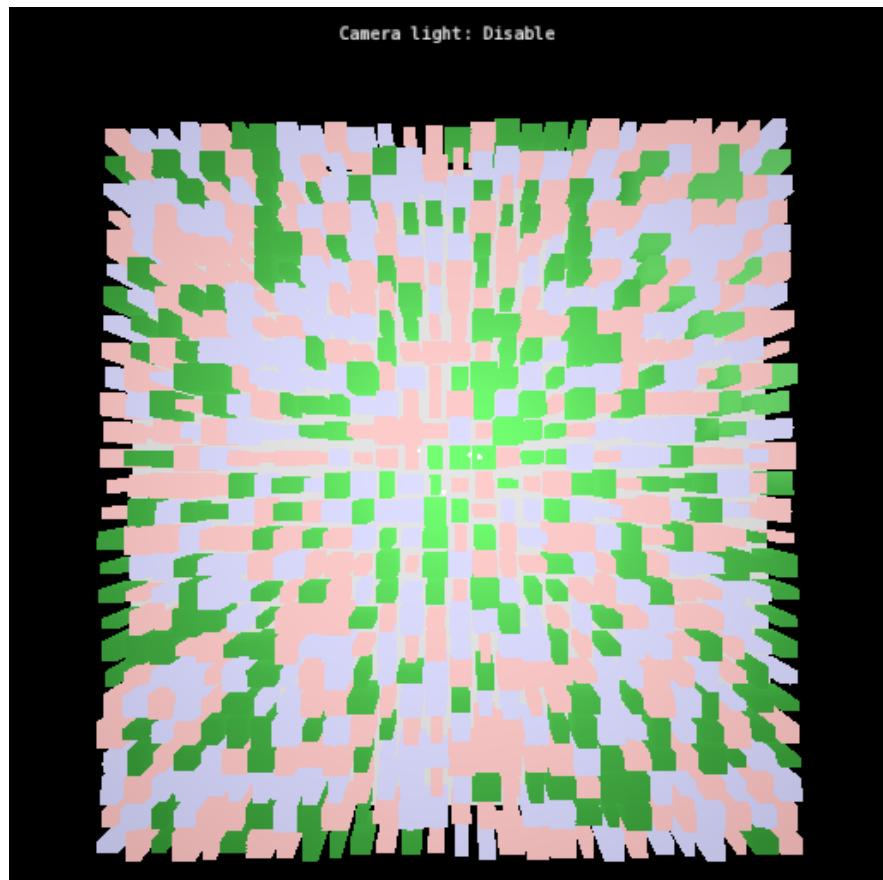


Fig. 21: Objects Placed following a noise function

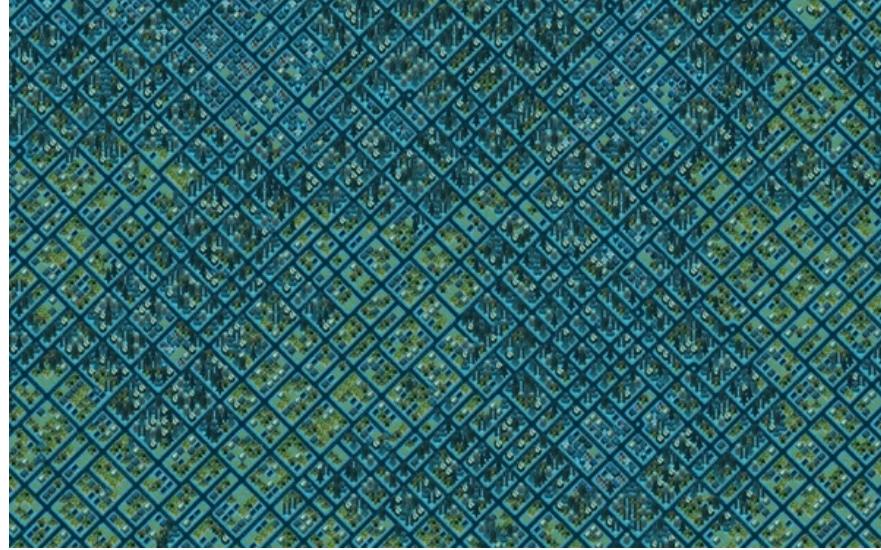


Fig. 22: Objects Placed with a noise plane from [1]

2.2 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 manny 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”. [15]

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 pipline(Vertex Data) in Figure 23.

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 tesselated and increase of the number of primitives to improve the models dynamically.

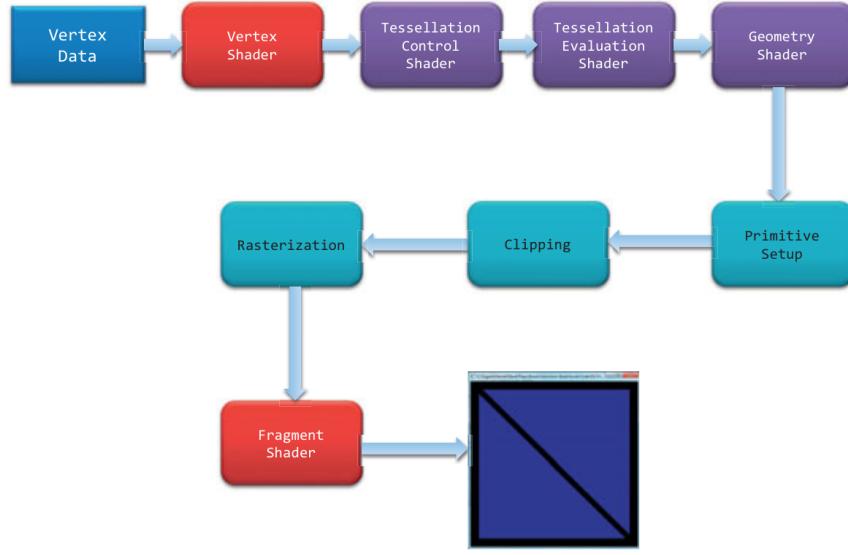


Fig. 23: OpenGL Pipeline [15]

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 assemble the vertices into primitives, clip the geometry cutting the parts that falls off the “screen” and the generation of fragments respectively.

A fragment is a “candidate pixel”, in that pixels have a home in the frame-buffer, while a fragment still can be rejected and never update its associated pixel location” [15].

2.3 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 perform computations to calculate the position of the vertices in screen coordinates, assign vertex’s color using lighting 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.

2.4 Tesselation Shaders

Tesselation Shaders are very different from the previous ones. These shaders address some of the limitations presented before. These shaders work with a geometric primitive called a *patch*.

2.4.1 Patch Are lists 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.

2.4.2 Tesselation Control Shader This shader generates the tesselation output-patch vertices and specify the tesselation level factors.

The output-patch vertices are the list of vertices that results after the input vertices have been processed.

The tesselation level factor defines how much the output patch is tessellated.

OpenGL supports three tesselation domains: a quadrilateral, a triangle, and a collection of isolines [15]. To control the amount of tesselation two sets of values are set, the outer-tesselation values and the inner-tesselation values. These values define how the perimeter or the interior of the domain are subdivided respectively.

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

2.5 Geometric 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 than the input and output more vertices than the input. Also in these shaders the primitives type can be modified, i. e. the input can be *quads* and the output be a *triangle_strip*.

2.6 Fragment Shaders

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

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

2.7 DirectX

Graphics API made by Microsoft to their platforms, both Windows OS and their XBox Gaming platforms. This API is only used with these platforms therefore is not compatible with the core concerns of Rosetta that is portability.

3 Objectives

The primary goal of this work is to provide an alternative for the rendering of large amounts of geometry as fast as possible. This system should also support a significant amount of geometric primitives to be able to create the larger set of models as possible.

This work is being developed in the context of a bigger project, the Rosetta Project that aims to create a tool for generative design to help architects and designers to develop their work procedurally.

In the context of Rosetta this system will act as a preview mode that allows the users to rapidly see the changes they make on their model during their creative process.

The main focus is to implement a set of geometric primitives such as *boxes*, *cylinders* or *spheres* that will allow users to model and some example functions such as *city*.

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

4 Related Work (17pgs)

4.1 CityEngine [12] [11]

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 of 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 has a default value, but 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.



Fig. 24: Road Map growth

The Figure 24 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.

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”[12]. 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 ??.

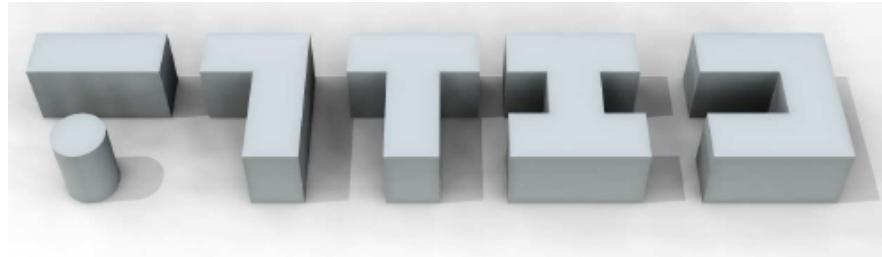


Fig. 25: caption

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.

4.1.3 Cities The result can be an city like Figure 26, with approximately 26000 buildings.

City Engine results can be imported by Maya, to achieve better results. Like the Figure 27, that represents a ‘virtual’ Manhattan.

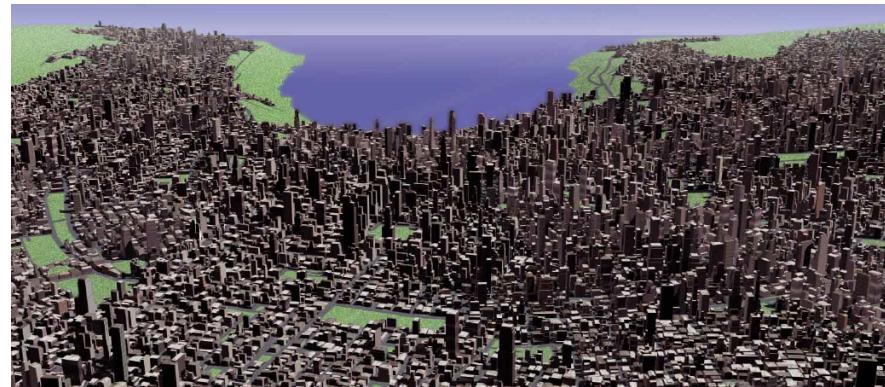


Fig. 26: City with approximately 26000 buildings.



Fig. 27: City rendered with Maya.

5 Architecture

This work will follow the architecture described in Figure 28. 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.

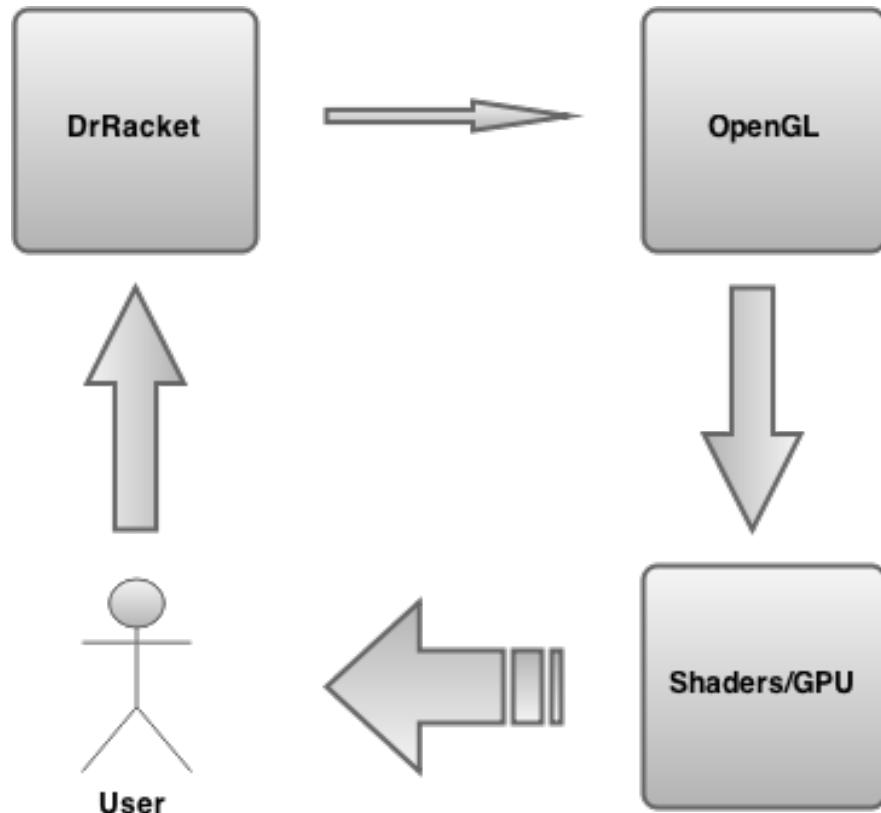


Fig. 28: High Level Architecture

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 embeded with an array of floats that encode the type of geometry and specific information like size or number of sides.

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. The

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 29 that is a procedural generated model of a city with 40k buildings.

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

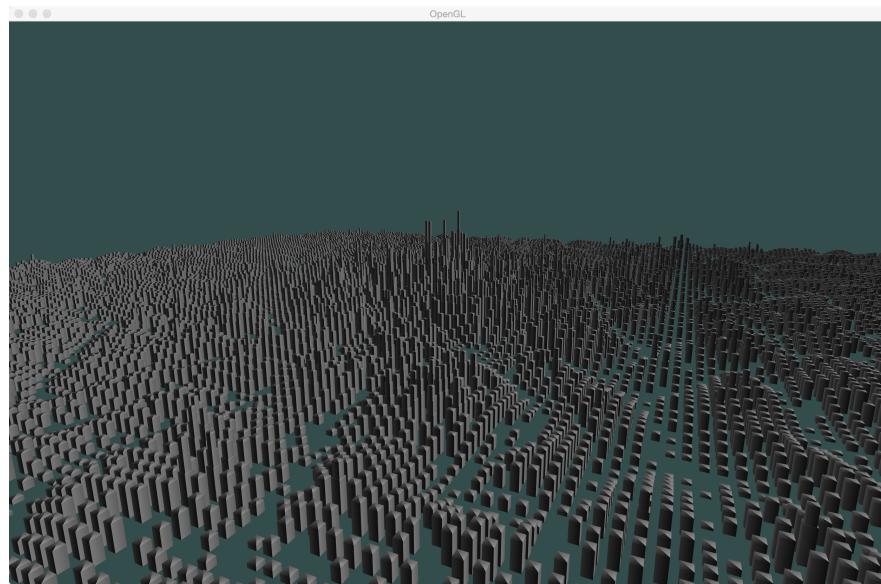


Fig. 29: City Example

6 Evaluation (1/2pgs)

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.

7 Conclusions

There is a need for this work because it as shown it tackles an important problem that architects and designers face and will face even more in the future. This system must have a good performance and suport the most used functions that the users need.

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

One prototipe is implemented that already creates boxes and cylinders without transformations that allowes the creation of the city example in Figure 29.

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. George Kelly. An Interactive System for Procedural City Generation. 2008.
8. George Kelly and Hugh Mccabe. A Survey of Procedural Techniques for City Generation.
9. 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.
10. Benoit B Mandelbrot, Dann E Passoja, and Alvin J Paullay. Fractal character of fracture surfaces of metals. 1984.
11. 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.
12. 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.
13. 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.
14. Daniel Shiffman. *The Nature of Code*. 2012.

15. 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.
16. Eric W. Weisstein. Tiling. From MathWorld—A Wolfram Web Resource.<http://mathworld.wolfram.com/Tiling.html>. Accessed: 2014-10-16.