2017-04-27

Urbis Terram - Designing and Implementing a Procedural City Generation Tool for Unity3D Game Engine

Yakin Najahi
Worcester Polytechnic Institute

Urbis Terram

Designing and Implementing a Procedural City Generation Tool for Unity3D Game Engine

By

Yakin Najahi

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

In

INTERACTIVE MEDIA AND GAME DEVELOPMENT

May 2017

APPROVED:

_____

Professor Charles Rich, Advisor

_____

Professor Dean O'Donnell, Reader

_____

Professor Mark Claypool, Reader

Abstract

The use of procedural content generation is becoming more and more popular in the video game industry. With games such as *Minecraft* or *No Man's Sky* we have seen the potential of PCG in video game creation but also its challenges. In fact, while the processing power and memory capabilities of our machines are unceasingly growing, human capability for content creation doesn't seem to be able to follow the same pace. Game developers had then to come up with several techniques and methods that will help them generate lots of content for their games while still keeping a certain level of control on the output.

Urbis Terram is a procedural city generation tool for Unity3D that allows the creation of complete cities to be used in video games or simulations made with this engine. The goal of this thesis is to tackle the technical challenge of designing and implementing a PCG tool that will help game developers to quickly generate terrains, road networks and allotment spaces for buildings and other urban areas. The goal is to have a unique complete city generation tool that can enable quick game design iterations and can be used to create complex virtual worlds.

Acknowledgments

I would like to express my deepest sense of gratitude to Professor Charles Rich for his valuable help and guidance throughout this master thesis. I am sincerely thankful for his support, encouragement and valuable suggestions that allowed me to pursue research on challenging topics that I am truly passionate about.

I am also very grateful to the committee members Professor Dean O'Donnell and Professor Mark Claypool for their understanding, patience and for giving from their time to review my work.

I would like to also thank all the participants in the user study sessions especially from the WPI IMGD community. I am grateful for their time and their valuable feedback that allowed me to quickly iterate through the project, fixing bugs and polishing the overall user experience of the tool.

Finally, I would like to also thank my family and friends for their continuous support and for always believing in me.

Contents

Table of Figures

Chapter 1. Introduction

1.1 Context

Procedural Content Generation in games or PCG is the algorithmic creation of game content with limited or indirect user input [1]. In other words, it is an algorithm that is able to create game content on its own following specified rules and designer inputs. Procedural content generation started being used in video games since the early games. Games like *Rogue* (see figure 1), *NetHack* or *Moria* are amongst the first games that used PCG to create worlds and quests since the nature of their gameplay mechanics rely on pseudo-randomness and replayability [2]. These games would not have existed without PCG due to the technical limitations of hardware at that time. Low memory and disk space along with slow CPUs made developers think of other ways to create content on the fly without the need of storing and loading data.

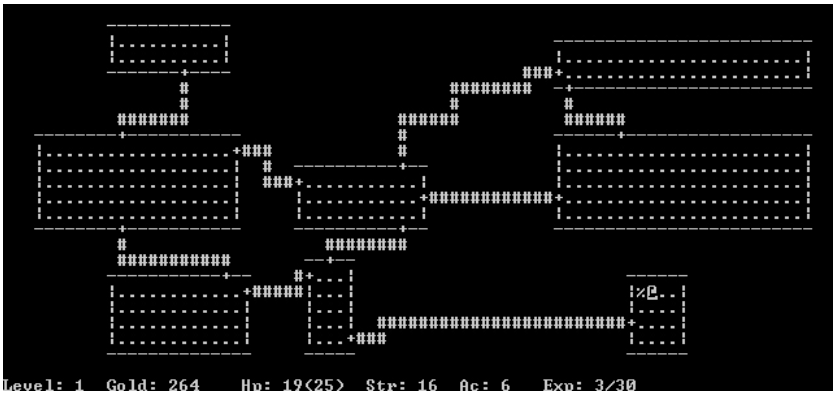

Figure 1 - Rogue: First popular game that used PCG

However, what started as a technique to counter hardware limitations soon became a way to create huge worlds in a short amount of time by controlling the procedures' parameters instead of hand making every single asset in the game and manually placing it. In fact, while storage capacity is becoming bigger and bigger and while processors are becoming more and more powerful, human

ability to create content cannot seem to follow the same pace. Recently, using procedural generation in video games has become a common practice among game developers. Creating content programmatically (3d models, textures, quests, etc…) allows the reduction of production time and also increase the replayability of the game. This allowed the creation of massive and almost infinite worlds like in *No Man's Sky* or *The Elder Scrolls: Daggerfall*, a game that has twice the size of the British Isles.

In this project, we are going to develop a Unity3D tool for advanced procedural city generation. We consider the algorithm we intend to implement as advanced because it will rely on fitness functions, feedbacks, designer input and other specified rules to generate world maps, roadmap systems, city blocks and also buildings and their placement in a realistic way, inspired by real life city layouts and organization. The goal of the project is to tackle the technical challenge of procedurally generating a coherent and organized city. The tool should allow designers and game developers to quickly generate terrains and cities for their games. It should also allow fast iteration and provide enough control on the output. Furthermore, it should be possible to use the algorithm in both online and offline settings: to generate worlds while using the editor and before building the game or to the generate worlds on the fly before the start of a game. Finally, while there are a lot of city building and management games, there seems to be no commercial games that use procedural city generation.

We will evaluate our tool based on three aspects: visual attractiveness, city generation coherence and lastly the efficiency of the algorithm. Designing PCG based games poses several problems. Several PCG games lack visual attractiveness with inconsistent designs. Other games lack coherence in their content generation due to the lack of designer control and algorithm feedbacks leading to improbable and unbelievable situations (e.g. final boss in the second room of

the game or a bedroom that shares a door with a prison in a dungeon). Lastly there are games that, while having impressive world creation algorithms, become very quickly repetitive or lack gameplay depth due to the abstraction of game design control during the development phase.

1.2 PCG in Games

While procedural content generation is still a very popular research subject, it has also been used in several video games for various purposes. *Minecraft* (see figure 2) is probably the most popular game in this genre [3]. The game populates the game world with terrains, caves, water flows, monsters, etc… using several techniques like Perlin noise for terrain creation or Whittaker diagram [3] to divide the landscape into biomes. Both of these techniques will be used in our project.

Figure 2 - Minecraft: One of the most selling games of all time used PCG for world creation

Other popular games using PCG are *Spelunky* (see figure 3) and *Diablo*. These games use procedural content to propose a fresh and new experience for the players each time they repeat the game or the level. The player will never experience the same dungeon or the same level layout. These are good example of games using PCG to create experiences that can be played several times while still renewing the challenge by creating unpredictable levels. The player can't

memorize the enemies or the items placement and thus the challenge and the sense of discovery is refreshed [4].

Figure 3 - Spelunky: PCG for infinite cave creation

Another type of games that use PCG extensively are Real Time Strategy games. Games like *Civilization, Age of Empire* or *Warcraft 3* have procedurally generated maps in their multiplayer and skirmish modes. These maps are particularly interesting since the players can't predict the location of the resources and have to explore the map to locate them while also trying to localize their enemies. This kind of maps increase the sense of danger and challenge, making games more interesting and fun. However, this can lead to unbalanced games especially in strategy games. A player can have an easier access to certain types of resources or can start the game in a more advantageous location. Procedurally-generated maps promote exploration, but they are very difficult to make balanced for different play styles and strategies [4].

The most popular game released recently that uses procedural content generation is *No Man's Sky* (see figure 4). This game uses PCG as a core gameplay mechanic and the whole game is generated from a simple seed allowing the generation of an almost infinite world with billions of planets and galaxies [5]. This is a perfect example of the immense possibilities that PCG offers. However, while being technically revolutionary, *No Man's Sky* received bad reviews and had an

overall poor reception from critics for its repetitiveness and lack of clear purpose. This gives us an insight on the power of PCG but also its design challenges.

Figure 4 - No Man's Sky: Most recent popular game released using PCG to generate an infinite world of planets and galaxies

Chapter 2. Background and Related Work

2.1 Terrain Generation

The terrain is the most important piece of the scene we're trying to create with our tool. The terrain is going to dictate all the next generation steps therefore it should be the first part we generate. Several research studies have been done on how to create, manipulate and visualize terrains. In this section, we're going to enumerate and describe some of the major techniques and data structures used to define and create terrains.

In his paper entitled Real-time Procedural Terrain Generation [7], Jacob Olsen describes a way to generate a base terrain using "pink noise". This type of noise is characterized by the spectral energy being proportional to the reciprocal of the frequency $P(f) = \frac{1}{f^a}$. This method is very common to generate basic terrains in computer graphics applications and one major technique to generate this type of noise is to use midpoint displacement method. In [7] they chose the diamond-square algorithm as an implementation for this method to compute the noise values for the generated height map. The algorithm takes a square matrix of size $2^n + 1$ to recursively compute halfway values by averaging the corners and applying an offset to the result. The algorithm is an alternation of two phases: the diamond phase and the square phase (see figure 5).

- Diamond phase: the center of each square is the average of the four corners of the square with the addition or the subtraction of a random offset that decreases after each step.
- Square phase: the center of each newly created diamond is the average of the four corners of the diamond with the addition or subtraction of a minor random offset that decreases after each step.

- The iteration step is then halved and the process is repeated.

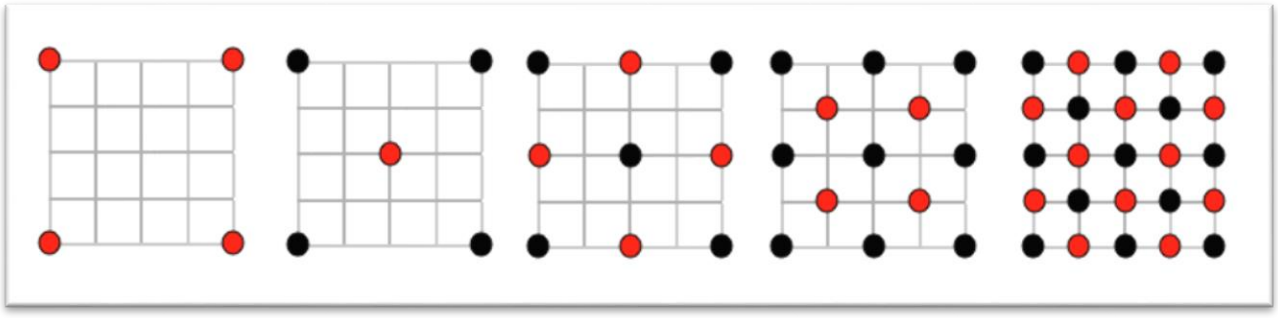

Figure 5 - Diamond Square algorithm steps

They finally combine the result with a Voronoi diagrams based noise to create a base height map and then use different types of erosions to create the final terrain. This is one of the two methods we implemented for the terrain generation tool as described later in chapter 3.

Another method to generate terrain by using noise generated textures as height maps is described in [8]. Perlin noise in particular is used in a lot of map generation algorithms since it provides a good distribution of height and gives a good approximation of a map. It is a coherent type of noise that represents the sum of several coherent noise functions of ever-increasing frequencies and ever-decreasing amplitudes.

Perlin noise generation follows these three steps:

- Grid definition of size n with random gradient vectors. At each node of the grid pseudo random vectors are chosen randomly from the unit circle.
- Dot product computation between distance and gradient vectors. The distance vector is the distance between each point and each corner point of its node.

- Interpolate the dot product values between nodes. The interpolation function should have a zero first derivative at the 2^n grid nodes. This prevents gradients generated during the definition phase on these nodes from changing conserving the noise map consistency.

Despite its pseudo random look, height maps generated using Perlin noise have details of similar sizes and amplitudes. This property allows the texture to be easily tunable with parameters. The most common parameters for Perlin noise are octaves, frequency, persistence and lacunarity:

- **Octaves:** each coherent noise function that we use to generate the final Perlin noise is called an octave. Same as in music, each successive octave should have the double of its predecessors' frequency. The more octaves we use the more details are going to be added to the final result but also the bigger is the calculation time.
- **Frequency:** frequency of each octave. It usually represents the rate of appearance of bumps and details.
- **Persistence:** the persistence is a multiplier that specifies how quickly should the amplitude between successive octaves drop. Increasing the persistence value tends to make the generated terrain more rough.
- **Lacunarity:** lacunarity represents how quickly the frequency should increase between successive octaves. It's common to use a lacunarity of 2 to obtain the same octave property of music.

Using Perlin noise to generate height maps was our primary way to generate terrains in Urbis Terram, its good performances and the fact that it is highly customizable with parameters made this method particularly effective in our generation process as described in chapter 3.

In [7], they propose a method of generating terrain using procedural models based on hydrology where the map creation is done through physics based algorithms that uses water flow data. Starting from a user specified terrain contour, river mouths and set of rules the algorithm generates a terrain by creating a river network graph and simulating the effect of the water flow on the landscape. The algorithm follows these steps to generate the final terrain (see also figure 6):

- User specified contour, river mouths and control parameters.
- Generate a river network graph
- Classify the river patches and isolate the data using Voronoi diagrams
- Generate the final terrain landscape using a predefined terrain construction tree

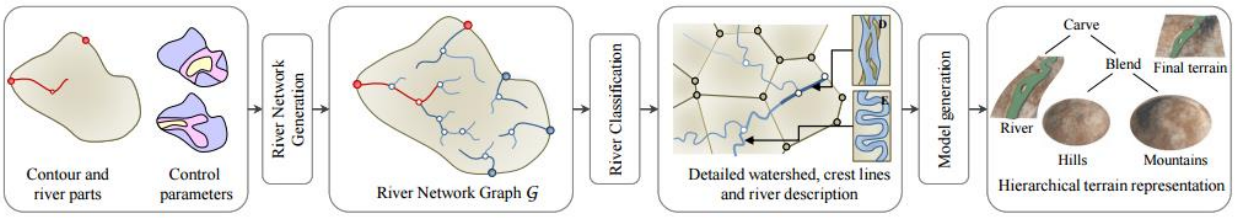

Figure 6 - Overview of the terrain generation method used in [7]

While this method is very interesting and can produce realistic looking terrains that simulate water flow and erosion, we decided not to implement it in our project due to its complexity and for being out of scope.

2.2 Road Network Generation

The second step in the generation process would be the procedural city generation where the goal is to generate a city that takes into account the environment and the height information of the map. By city we mean the road network system and the lots/subplots, parcels/sub parcels system. In [9], George Kelly and Hugh McCabe present a system called Citygen used for generating cities by dividing the problem into three stages: Primary road generation using adjacency lists,

Secondary Road generation using L-Systems and then building generation using geometry subdivision.

For the primary road system, they use a two graph system to store the data required to generate and visualize the roads, a high level graph and a low level graph (see figure 7). These graphs represent adjacency lists describing the interconnection between the different nodes. The high level graph holds information on primary roads intersections, thus storing the topological structure of the primary road network. The low level graph describes more in detail the path that follows the primary road from one node to another. The nodes from the high level graph can be manipulated manually by the user.

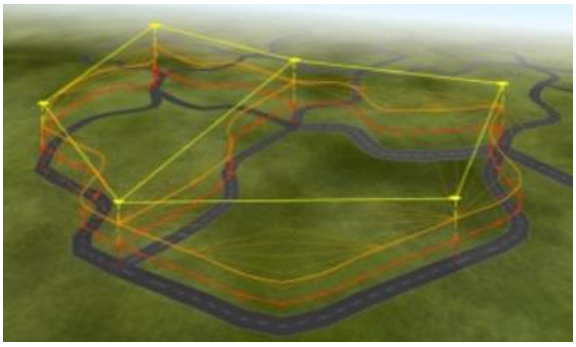

Figure 7 - Primary road network system. High level graph in yellow, Low level graph in red

For the secondary map network, they consider the region enclosed by primary roads as city cells and they generate the secondary roads inside of these cells. A parallel growth based algorithm is used to create the secondary roads using L-systems (figure 8). The secondary roads are gradually generated in a parallel fashion starting from the cell borders. Proximity and intersection checks are being done along the way to correct and fix possible connection and visual artefacts.

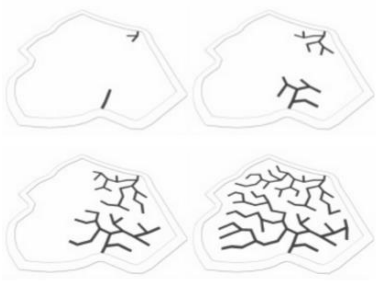

Figure 8 - Secondary roads generation steps as described in [9]

Yoav Parish and Pascal Muller describe in [11] a similar solution for cities generation where the work is subdivided on smaller problems: Roadmap, allotments, and buildings generation. The roadmaps are generated using L-Systems and different road patterns while the lots are created using a scaling and intersection method. The particularity of the algorithm used here is that the L-Systems are based on how blood vessels look. The roads rarely end in dead ends and should either be connected to other roads, to other intersections or loop back to themselves. If a dead end is found a local check is performed on its surrounding area to find possible ways to attach the street to the road network (see figure 9).

- If the street intersects with an another street, an intersection is created and the street is shortened to end at the intersection.
- If the street ends close to an existing crossing or intersection, extend the street and attach it to the same crossing.
- If the street ends close to intersecting with a preexisting street, extend the street to create a new intersection with the already existing street.

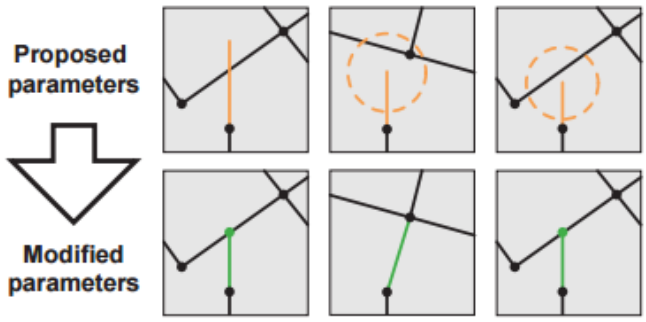

Figure 9 - Different methods to fix the dead end problem as described in [11]

In [12], a different approach is proposed for cities generation where they use urban land use models. The goal is to have a coherent city that looks realistic and is adapted to the terrain. They use urban data from different popular layouts (see figure 10) to guide the generation and then use a district placement algorithm to create a metropolitan city with a specified style (North American, European, etc…). European cities tend to have a historic center, surrounded by residential areas for the different classes (high, middle and working class) and then we'll find industry focused areas in the city's periphery with the heavy industry usually located near the water to facilitate transportation. However north American cities usually have business district or "downtown" which represents the city core. That's where we see most of the skyscrapers and large buildings. This core is then followed by a lower density surrounding core that contains housing, warehousing, medical and education facilities. Then we'll find the malls, the manufacturing and large industry areas outside the city's perimeter.

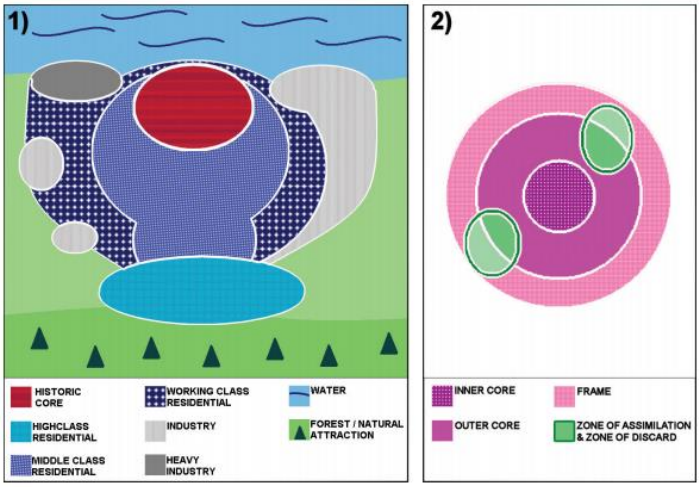

Figure 10 - Example models for a European city (left) and American city (right) [12]

The generation method used in [12] to create cities also follows three steps (see figure 11):

- Setting up the basic parameters of the city. Size, location on the map, layout style (European, American …), number of districts or the number of highways are some of the parameters that can be set during the initialization phase.
- Then the preliminary highways and candidate locations for the districts are computed and found inside of the city limits.
- Finally, the districts are generated and placed inside the city limits following the layout style used. The streets will then be generated with different densities and layouts depending on which district they are included.

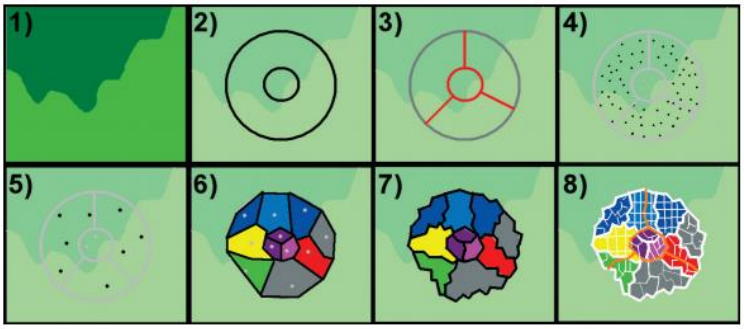

Figure 11 - City generation process as described in [12]

In our project we're going to use a mix of these solutions. Dividing the road map generation into two steps, primary and secondary and using L-Systems is an efficient way to create blocks while keeping a certain degree of control. Also as described in [12] the city needs to be adapted to the environment and be built in a believable way. The algorithm might need to iterate several times through the city to achieve the realistic and coherent layout needed.

2.3 Allotment and Parcels Generation

After the terrain and road network generation systems are done, the next logical step is to subdivide the city regions into plots and subplots and then find locations where to generate buildings inside of these regions.

In [11], Yoav Parish and Pascal Muller describe a recursive solution to create plots. They consider every closed loop of streets as a unique region called "block". Then they run a recursive algorithm on these blocks to subdivide them into lots. The algorithm detects at each step the two longest sides that are approximately parallel and subdivides them using a perpendicular segment. This creates at each step two new subplots per lot. The algorithm stops when the subplots area goes below a specified threshold where they are considered as being too small to hold buildings. The result of this type of subdivision should result in all convex areas if it starts from a convex shape.

A similar method for lot subdivision is used in [9] but with a major improvement. Instead of limiting the subdivision process to only work with convex shapes, they extended the algorithm to make it work with concave shapes (see figure 12). This has the advantage of working better in suburban areas where blocks tend to have more abstract shapes unlike urban areas where blocks

have a more rectangular look. The algorithm subdivides each region recursively by looking for intersection points between the perpendicular to the longest side and the region sides. Sub regions are then created from the previous shape and using the newly found intersection points. If the sub regions are still bigger than a threshold area, the algorithm is applied again.

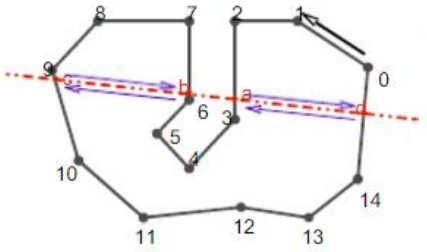

Figure 12 - First step of the subdivision algorithm on a concave shape as described in [9]

This was one of the two methods we implemented for allotment subdivision. We used the oriented bounding box algorithm to perform this type of subdivision. The algorithm is described in section 3.31. The fact that it works for different types of shapes, even non convex shapes makes it extremely reliable and would serve as robust first step to create sub parcels for buildings and other urban areas.

Another approach for allotment is described by Carlos Vanegas et al. in their paper "Procedural Generation of Parcels in Urban Modeling" [10]. They propose a skeleton-based subdivision that creates parcels similar to north American parcel subdivision where the front of the parcels is along a street and the rear side is adjacent to other parcels. They start by collapsing the polygon (see figure 13) to define the possible space for the parcels. An offset distance from the streets to the rear of the parcels is defined before the collapsing method. If the offset distance is small, an interior region with no access to roads will appear and have to be ignored in the future. However, if the offset distance is sufficiently big, all the generated parcels will be in contact to each other.

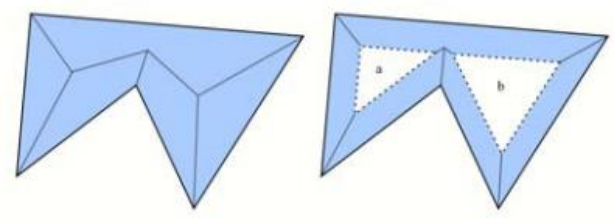

Figure 13 - Initial Collapse of the polygon shape and removal of triangular strips

This creates what they call strips, or regions that are parallel to roads where parcels will be generated. If a strip has a triangular shape, its edges will be collapsed and merged to its adjacent strips. Finally, parcel subdivision will be applied on these strips by linking the roads to the rear edge of the strips as shown in figure 14.

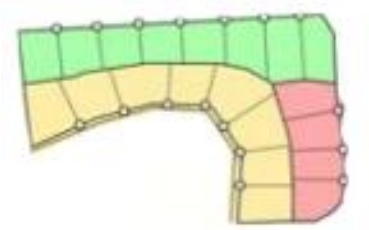

Figure 14 - Sample of a parcel subdivision obtained using the skeleton-based subdivision algorithm

We didn't implement this type of subdivision in our project due to time limitation and scope. However, we are planning on adding it for future iterations of the tool since it produces different types of parcels that can mimic certain types of real life areas.

2.4 Procedural Content Generation Tools

We looked at several existing PCG tools for references but also to make a competitive analysis. We started with general purpose commercial tools.

Substance Designer: made by Allegorithmic, this tool allows the creation of procedural textures for all types of computer graphics applications including games. It is a node based tool

with a vast library of functions that can be combined to simulate several effects like rust on metal or plastic shininess. It's an industry standard and is commonly used in AAA companies.

SideFX Houdini: another industry standard tool, Houdini is an extremely powerful used in video games but also in the cinema industry and allows the creation of 3D assets, procedural 3D animations, special effects and particle systems to name a few. The particularity of Houdini is that developers can extend its functionalities and create tools inside of it to make it fit their needs.

CityEngine: created by Esri, CityEngine (see figure 15) is a tool with a more specific purpose than the ones we just mentioned. This tool is primarily used for the procedural generation of cities. It is used as a visualization software for city planners and architects. It also offers the possibility to generate cities using Geographic Information Systems (GIS) data. While this tool allows the creation of relatively realistic looking cities, the output can't be used inside game engines.

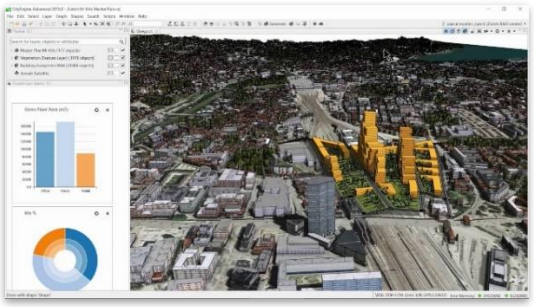

Figure 15 - Esri's CityEngine

We also looked at PCG tools that existed on the Unity asset store and that were closely related to our project.

Procedural Terrain Generator: this is a single purpose tool that focuses on procedural terrain generation. It's also based on height map generation using noise maps. However, the results are of low quality (see figure 16) and don't look realistic, also the tool has a really high price range.

Figure 16 - Procedural Terrain Generator (Unity3D tool)

Simple Cities – Procedural City Generator: is another single purposed tool but this time focused on cities generation. As we can see in figure 17, this tool generates only cities in a grid which doesn't look realistic. Also the cities generated are generic and don't adapt to the terrain. The price range is also high which another opportunity for *Urbis Terram*.

Figure 17 - SimpleCities - Procedural City Generator (Unity3D tool)

World Creator Professional: probably the most professional looking tool we found on the unity asset store that was related to our work. This tool allows the creation of realistic looking terrains (see figure 18) with a high quality output. World creator is capable of creating beautiful

scenery and we would certainly use it as an inspiration during our development process. We plan to differentiate ourselves from this tool with our complete city generation features and also with a much lower price range.

Figure 18 - World Creator Professional (Unity3D tool)

Investigating these tools and analyzing their functionalities and outputs gave us a lot of inspiration for *Urbis Terram.* Successful and commercial tools such as HoudiniFX and Substance Designer are both very polished and provide a highly pleasant user experience by presenting all the functionalities through a clear UI and by using node based programming. While creating a node base system for our tool is out of scope during this master's thesis, it's definitely a feature that we might want to add in the future. In addition to that, we noticed a real lack of city generation tools on the unity asset store. Most of the tools are single purposed (primarily for terrain generation) and even the ones that try to generate cities, have a poor nonrealistic output. This is an opportunity for us to create something new for the asset store with more functionalities and also with a more competitive price.

Chapter 3. Design and Implementation

The purpose of our project is to create a tool capable of procedurally generated content to create world maps (terrains or planets) and populate them with cities. These cities will have a road map network but also lots, subplots and buildings. The goal is to have a coherent city that populates the map in an efficient and realistic way.

3.1 Advanced Procedural City Generation Tool

As mentioned above the goal of this project is to tackle the technical challenge of procedurally generating a coherent city and how to design a game based on it. We described our intended algorithm as "advanced" since it needs to be able to create a coherent, complex and realistic city. During our research for inspiration sources we noticed that there is a lack of games based on procedurally generated cities. PCG in games is commonly used for creating world maps, dungeons, weapons or stats, but when a game involves cities, it's either a city premade by designers and artists (e.g. Grand Theft Auto) or a game where the goal is to build and manage cities (e.g. SimCity, Cities Skyline). Most of the city generation algorithms we encountered are for research purposes or for a modeling/architecture software [14]. The problem with most of these algorithms is that they propose a way to create a rough generic city by creating a road systems and populating empty spaces with buildings in various sizes. For visualizations purposes this can be enough but when investigated closely, these cities do not always look realistic. For example, skyscrapers should be more concentrated in downtown area and a downtown is more likely to be located near a river or a lake. Also every city needs some common locations and buildings like a main square or plaza, police stations and fire departments in every neighborhood, a hospital in an easy access location, roads should be made on flat surfaces if the choice is possible, etc. Another feature that

city generation algorithms we encountered do not have is the ability to interact with the city, modify it or alter it. This feature is particularly important in our game. Buildings should have different representations and a flexible data structure should be used to store the information about the different buildings. This will make the interaction with a particular building, or buildings with the same type easier (if we want to swap the 3d models of a damaged building for example).

Our goal then is to create an advanced procedural city generation algorithm that tries to recreate a human-like city while still keeping a certain degree of control so that the designer can tweak the result. The final result is an intelligent living city that the player can interact with. To do so we are planning to implement several small modules, each of which is managing a certain step of the creation process where the output of one module will be used as the input of the next module. Based on fitness functions and procedure feedback, the algorithm will check the degree of coherence of each step's output before moving to the next step.

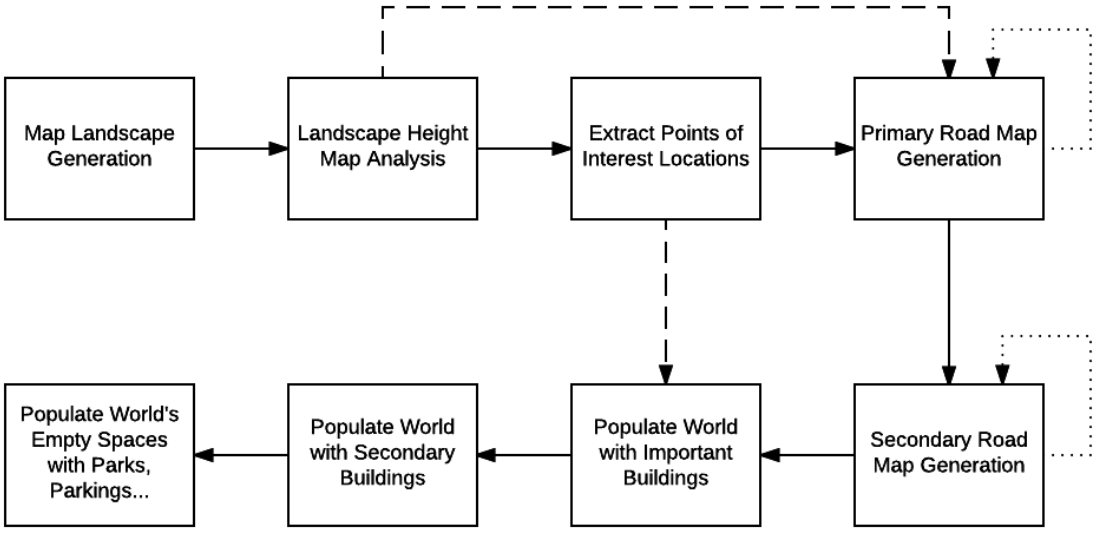

Figure 19 - Flow chart of our advanced city generation algorithm

Figure 19 shows a flow chart of the algorithm. We will start by a map generation module where we will generate a height map using Perlin noise and use that data to create a mesh that will be our basic terrain. The result will then be analyzed in the second module to be subdivided into different biomes (river, lake, hills, mountains, beaches, etc.). After the map analysis we're going to extract the points of interest in that map, for example the largest flat area close to a river will be the downtown and the second largest area will be an airport, etc. Then we move to our primary roadmap generation module which uses the output from the previous module but also information from the landscape analysis. This will allow us to generate big roads that interconnect the points of interest but also that take into the account the topology of the terrain. The next step is the secondary roadmap generation, where we will use L-System to create smaller roads in the areas between the big roads. This will result in city blocks that we can later use for buildings, parks, etc. The primary and secondary roadmap generation will use a feedback system based on a fitness function, so the algorithm will iterate until it finds an acceptable outcome. Using the city blocks information and the points of interest we can then start populating the world with important buildings and then move the secondary buildings. Finally, any empty space will be transformed to a park, parking lots, etc.

3.2 Terrain Generation Tool

3.2.1 Height Map Generation

The first step for our project is to generate world maps. The world map is the terrain that will hold the cities on top of it. The terrain can take the form of a patch of land or an isolated island. It should be relatively easy to use the generated terrain to create planets by combining several generated patches.

The Unity3D terrain system allows the possibility to import a height map to generate the terrain topology. A height map is a matrix of values between 0 and 1, zero being the lowest possible elevation for a point on the terrain and 1 matches the maximum elevation of the terrain. If we map these values to grayscale colors, we obtain a grayscale 2D texture that represents the topology of the terrain (see figure 20).

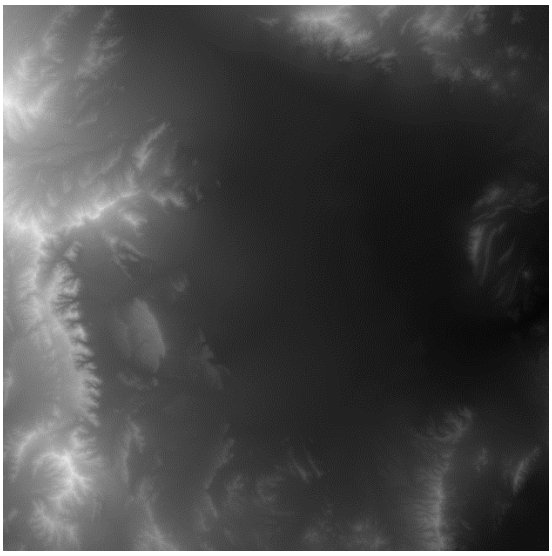

Figure 20 - Example of grayscale texture representing a height map (Las Vegas in this case)

Knowing the size of the terrain, it's relatively easy to generate a height map and map its values to the terrain points. However, if we used a generic random generator function to generate the texture's values, we would obtain an unusable noise map that resembles the white noise we see on TV screens. What we need is a noise function with a gradient output and a smooth distribution.

From the researches we made, we noticed that the most common used type of noise for this kind of applications is the Perlin noise. Invented by Ken Perlin in 1983, this type of noise is particularly useful in various application of computer graphics and allows the generation of several textures such as fire, clouds, marble patterns but also can be used to approximate terrain elevation

by generating height maps. Perlin noise has fractal properties and allow the generation of almost infinite maps, in number and in size. The advantage of the Perlin noise is that it is very modular and can be tweaked using various parameters like number of octaves, persistence and lacunarity. One more advantage is that the "Mathf" library included with Unity3D has a "PerlinNoise" function that returns a value between 0 and 1 given two coordinates x and y. If we tweak the indices of our matrix and pass them to the function to populate our 2D texture, we can quickly generate a usable height map.

We implemented a "GenerateNoiseMap" function to generate our height maps. The function takes the following parameters:

- **Width** and **Height:** control the size of the noise map
- **Seed:** the random generator seed value. This allows us to generate the same terrain if we use the same parameters and seed
- **Scale:** controls the zoom level on the height map.
- **Octaves:** controls how many iterations of Perlin noise we're going to use to generate the height map. A higher value generated a map with more details.
- **Persistence:** controls the amount with which the amplitude of the noise value is going to increase or decrease between each octave.
- **Lacunarity:** controls the amount with which the frequency of the sampling value is going to increase or decrease between each octave.
- **Offset:** allows the possibility to scroll through the Perlin noise generated height map.

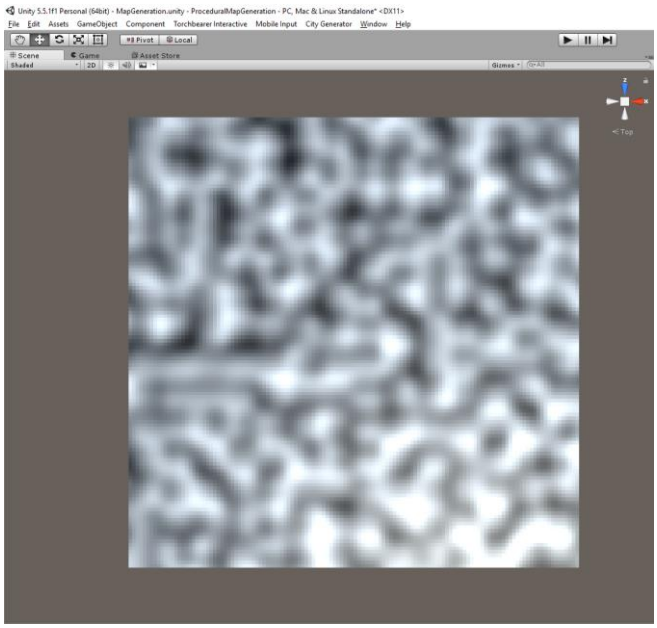

Figure 21 – Height map texture generated using our Perlin noise function in Unity

Now if we use the generated texture (see figure 21) as an input for the Unity terrain system, we can see the terrain's topology take shape. Various elevations which look like hills, plains and mountains will be visible on the terrain (see figure 22) and will follow the same pattern generated by the Perlin noise.

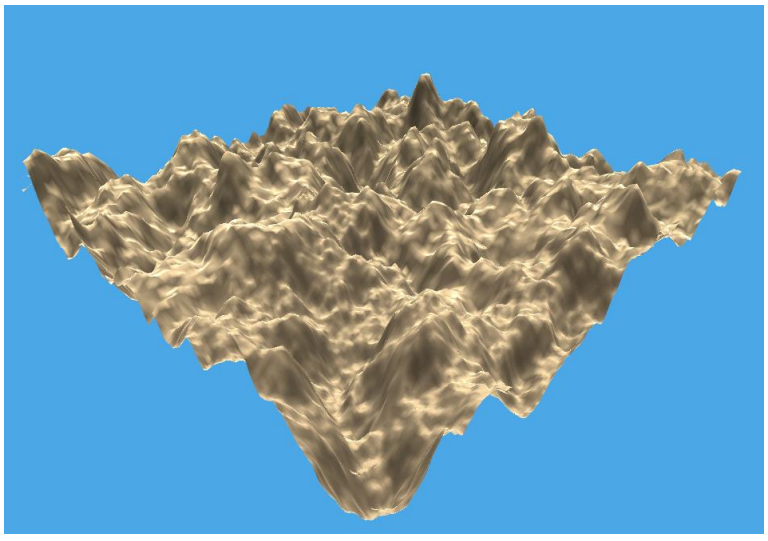

Figure 22 - Unity terrain with the Perlin noise map applied as height map

One more fractal method we used to generate the terrain height map is the Diamond-Square algorithm. As described in Chapter 1 section 2, this algorithm also allows the creation of relatively realistic looking terrains. Starting from 4 random height values at the corners of the map, the algorithm will fill the 2D matrix by averaging previously generated values while adding or subtracting a random value, alternating between a diamond and a square pattern.

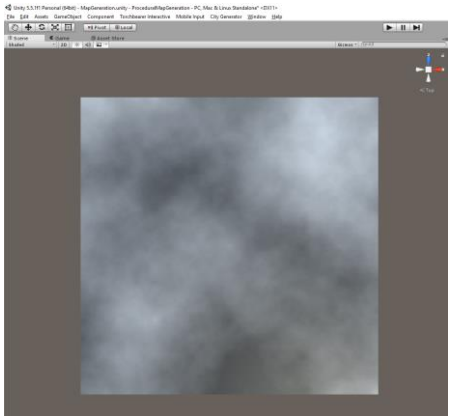

Figure 23 - Height map texture generated using our Diamond-Square function in Unity

Applying this height map to the terrain (figure 24) gives a relatively similar result to the elevations we obtained using the Perlin noise. Plus, the fractal nature of the noise shows smaller details on the surface if we zoom in (see figure 25).

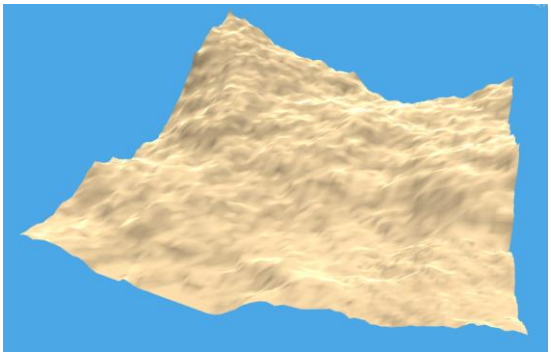

Figure 24 - Unity terrain with the Diamond Square noise map applied

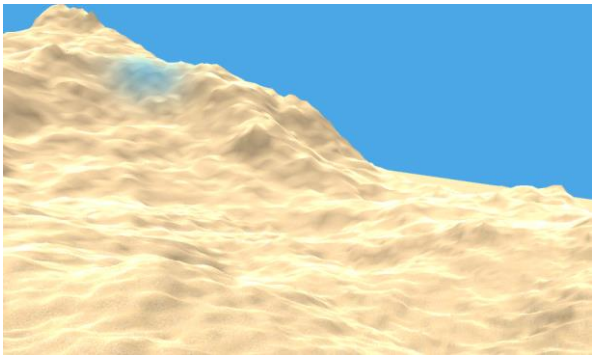

Figure 25 - The fractal nature of the noise shows similar details on the surface of the terrain

Because the Perlin noise function takes x and y coordinates as parameters to generate a noise value, it's possible to generate infinite data for terrains where it's possible to zoom in and out, offset the generated portion or generate other portions of the map on the fly. However, these properties are not available while using the Diamond Square algorithm since it generates the noise map based on a fixed size 2D matrix. Nonetheless, the latter method has the advantage of being less costly compared to the Perlin noise generator, especially if we use a high number of octaves to generate more details.

3.2.2 Biomes partitioning

Now that we have our height map and we applied it to our Unity terrain, we need a way to separate different the different biomes of our world. This is useful because it allows us to assign different textures and generate different types of vegetation for each biome. One way to do this type of partitioning is to use a 2D space portioning algorithm such as Voronoi diagrams. However, this doesn't work in our case since we need to execute this in local areas and have a consistent result across the map. Another way to do the biome partitioning is to use the height data from our height map. This has advantage of being relatively easy to implement since we already have the data we need. It's also easier to give control to the user to specify the limits of each biome.

We start by defining the different types of regions we plan to use to subdivide our map into biomes. For our project we decided to consider 4 types of textures: sand, grass, stone and snow (see figure 26). We assign lower or higher height limits to these textures and we compare these values to our height map to know what texture to use. If we choose to use lower limits, each limit works as the upper limit for the previous type of biome. So if we decide that grass should have a lower limit of 0.3, sand will have this value as its upper limit. Using this method is like painting the terrain from top to down.

If we consider that sand appears in undersea areas and on the beaches we can quickly notice that sand limits should be mapped to the lower values of the height map. Same with snow where we usually see it on top of mountains so it makes sense to give this type of texture a relatively high lower limit. Nevertheless, the user will still have the possibility to assign his own values and it's still possible to make a snowy map or a map with sand dunes.

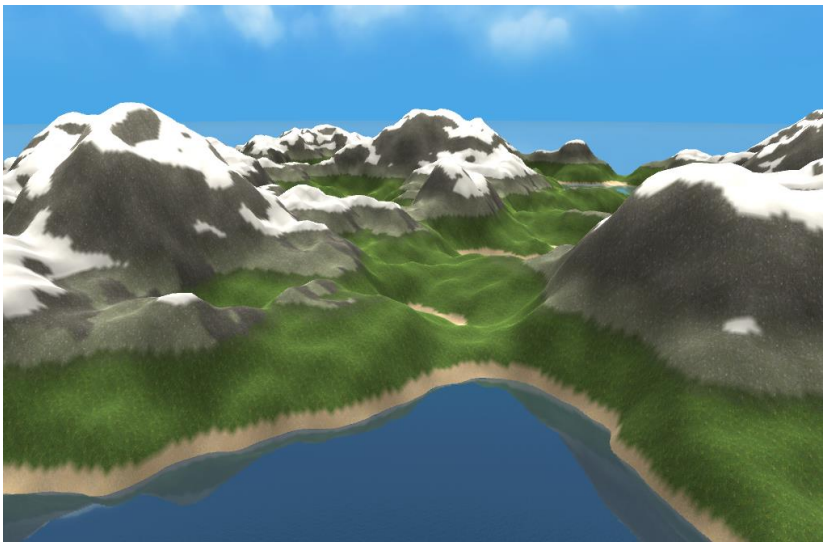

Figure 26 - Biome partitioning using height map data

Now that we have a way to define the limits of our biomes, we can use the same data to generate other types of details such as bushes, rocks, trees, etc… All we have to do is to take the same partitioning and use the Unity terrain API to apply these details with various densities depending on their location. For example, it's possible to generate high density bushes and trees on the grass areas, lower density bushes on the rock areas, no vegetation on the sand and snow areas. The result can be seen in figure 27.

Figure 27 - Using the biome limits data to generate vegetation and trees

It is also possible to use other types of information for the texturing and biome partitioning. For example, we can use terrain normals, steepness values and orientation to decide which texture to apply. The snow textures in figure 26 and 27 are applied on terrain areas with high altitude (high lower limits) but also with areas that have normals facing relatively up and with a low steepness value. This simulates how snow accumulation works in real life where it's more visible in flat, high altitude locations. All these parameters and values are tweakable to allow the user to create different types of terrains and environments.

3.2.3 Elevation adjustment

One final tweaking tool we provide the user with is the ability to adjust how the height map data is interpreted before being applied to the terrain. For this purpose, we created two interpolation

methods: the first is to smoothen the terrain's look and the second a falloff that give the map an island type look.

For the smoothing method, the user defines an interpolation curve that we will use to map the height map data to new values. For example, we can decide that the value of 0.1 on the height map is the sea level. If we want to have more plain areas between the sea level and the mountain areas, we can create a curve that remains relatively constant between the values of 0.1 and 0.7. By doing this, all the points that had an elevation value between 0.1 and 0.7 will have a similar elevation and thus giving the illusion of larger plain areas. We can also use this method to limit the height of certain biomes. We can for example decide to lower the height of the mountains but keep the other parts of the terrain in the same height. Examples of this technique are shown in figure 28 down below.

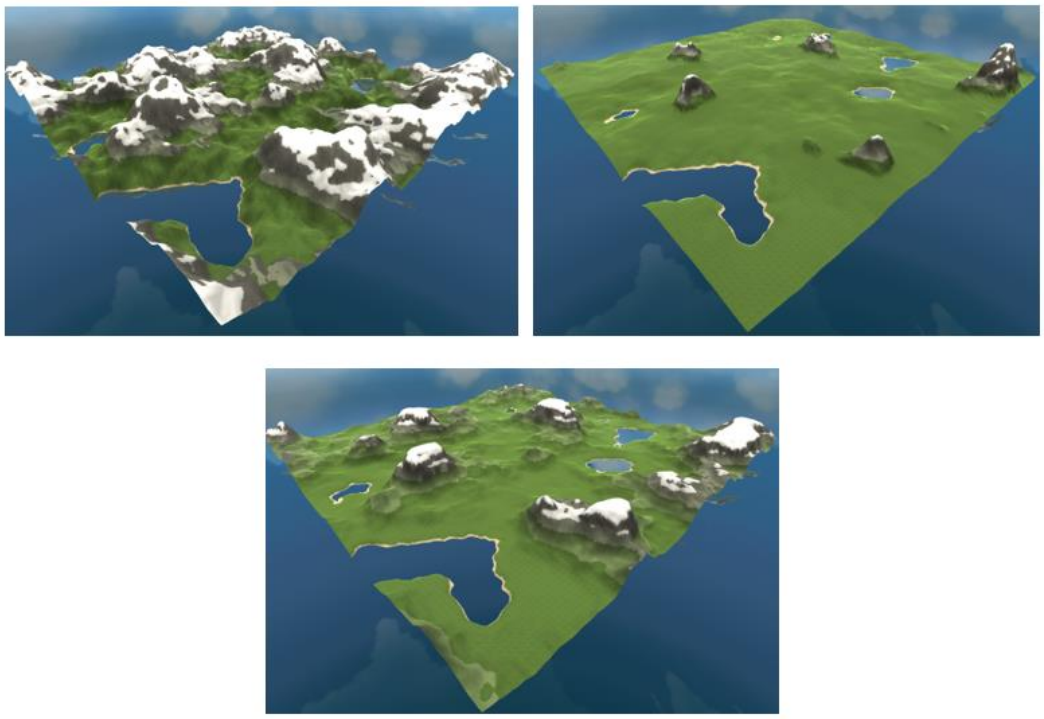

Figure 28 - Sample terrains with different smoothing properties (no smoothing, larger plain areas, same distribution of mountains and plains with flat mountains)

The second method for elevation adjustment is the use of a falloff map. We start by generating a 2D matrix representing a falloff that we will use to interpolate our height map data. To achieve an island look we need to nullify all the height map values that exist on the periphery of the map and gradually increase the value while going to the center. Using the following equation, we can interpolate each point in the map to obtain the desired island shape (see figure 29 and 30). Both ‘a’ and ‘b’ are tweakable values we can set as public variables in the tool’s inspector to control the falloff strength and speed.

$$f(x) = \frac{x^a}{x^a + (b - b * x)^a}$$

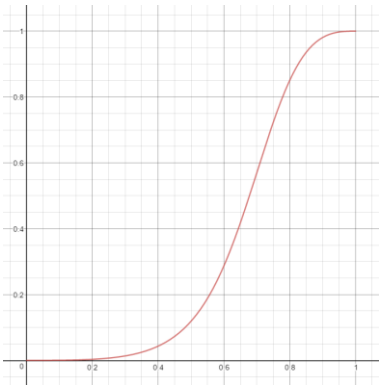

Figure 29 - Visual representation of f(x)

Figure 30 - Generated terrain before and after the use of a falloff map

3.3 Road Network Generation Tool

Now that the terrain generation component of our tool is complete, we can focus on the next step of the generation process which is the road network generation. We tried different approaches to generate primary and secondary road networks. Each method had its own advantages and disadvantages.

3.3.1 Voronoi diagrams

The first method we tried to generate road networks is to use Voronoi diagrams. Considering a series of random points, the Voronoi diagram creates cells around these points. To find the edges of these cells, segments are drawn between each two adjacent points. This segment needs to be perpendicular to the line between those two points and cross the line in its middle point. The intersection between the extracted lines define the end and start of the Voronoi cells' edges. Since it's a partitioning algorithms, it made sense to start by trying to use its properties to generate an interconnected road network (figure 31).

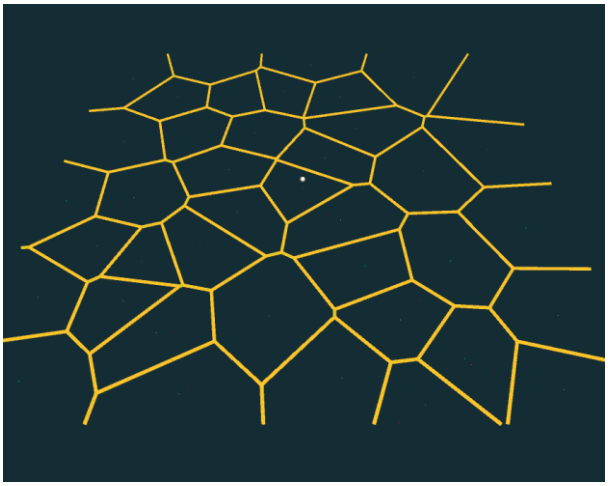

Figure 31 - Voronoi diagram cells generated from 40 randomly placed points

In figure 30 we see an example of a Voronoi diagram generated on Unity using 40 randomly placed points. While the result seems promising at a first glance, we quickly noticed that the result doesn't really represent real roads' layout. The cells are too irregular and have a very large number of intersections. Also these cells which might represent lots, have a relatively high number of adjacent streets: 6, 7 and 8 streets surrounding a single lot seems unrealistic and doesn't represent reality. We tried to fix this issue by generating a grid of points instead of random points to create the Voronoi diagram and offsetting them to create some irregularities. However, the results obtained are still not organic enough to feel like a real city network as shown in figure 32.

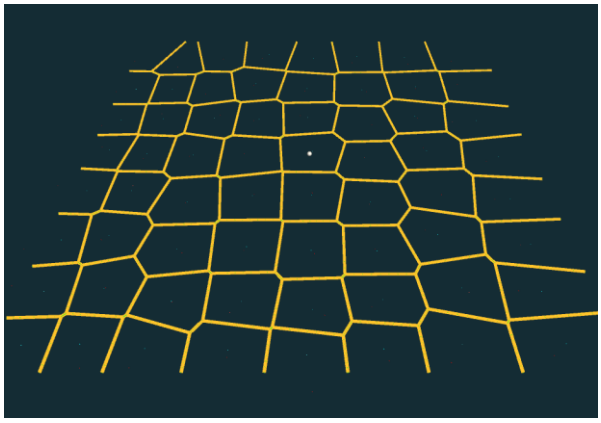

Figure 32 – Voronoi diagram cells generated using a grid of points

3.3.2 Visualization Recursion

The second method we tried is a method inspired by the visualization recursion technique used in computer graphics called Turtle Tail Drawing. In this technique we imagine a turtle moving in straight lines and that can rotate at the end of each line before continuing its path. While moving the turtle tail can be either down and thus drawing on the sand or up and therefore not drawing. This method of drawing is particularly used to draw fractal shapes, especially in shapes defined

through L-Systems. It's very easy to create a set of rules or a grammar that will dictate how complex shapes will be drawn. The tail up state represents the recursion nature of the algorithm where the drawing point can jump to a previous location where it branched.

Using this method, we created a generation algorithm for the streets that follow a recursion growth system. Incremental results are shown in figure 33.

- Starting from a central location, street segments will start growing in different directions.
- Each segment will trigger the generation of the next segment.
- The direction in which the segment is growing can change slightly by a random factor at each successive call.
- Each new segment has a chance to trigger the generation of a new street path perpendicular to the current one and thus creating sub streets.

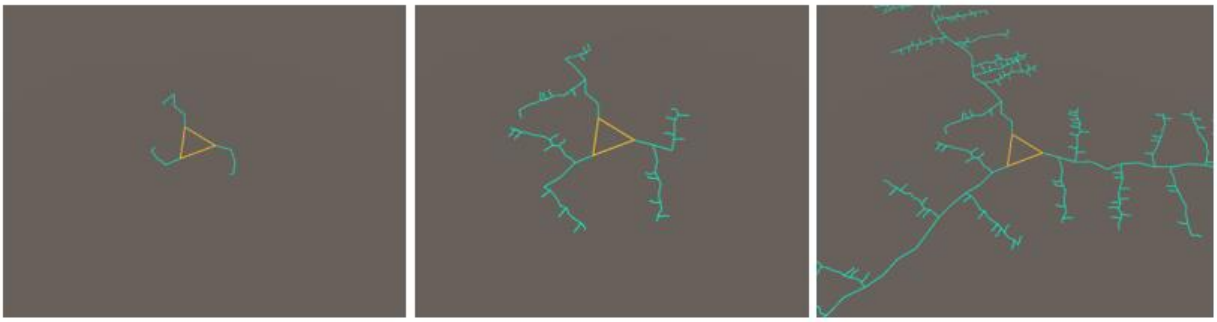

Figure 33 - Road network generated using the recursion system (left: start of the generation process, middle: first generated sub streets, right: end of the generation process)

Using this system, we have the ability to tweak several parameters to control the generation process:

- The number of starting streets at the beginning of the generation process.

- The number of road segments to generate.
- The angle variance between successive road segments.
- The angle variance when starting the generation of sub road segments.
- The road segment scale.
- The number of sub road generation iterations.
- Percentage (likelihood) for sub road generation…

This method has the advantage of being extremely modular and all these parameters give a lot of flexibility to the user to generate a wide variety of road networks. However, this technique has its own limits. The first disadvantage is that it doesn't have an intersection check due to the way it's generated. The recursion nature of this algorithm makes every street segment independent and it's hard to have a data structure that allows us to efficiently make intersection checks between all the road segments to fix them (see figure 34). One more disadvantage of this technique is that it doesn't create loops or closed areas which makes it harder to find lots and subplots for the buildings placement.

Figure 34 - Example of a complex generated road network where we see a lot of intersections happening

3.3.3 Space Colonization

The last method we tried for road network generation is a fractal technique inspired by A. Runions, B. Lane and P. Prusinkiewicz's paper "Modeling Trees with Space Colonization Algorithms". In this paper they describe a method for generating trees using a botanical growth system. Starting from a root point, they incrementally generate branches where their orientation is influenced by surrounding points called attraction points. If a space is filled by a branch, other branches will be generated in other parts of the space and thus creating an organic generation look. The algorithm follows these steps (see also figure 35) to generate a tree:

- Generate a series of random points (blue points in figure 34) on the space we want to colonize. The points will act as attraction points for the algorithm.
- Incrementally generate branches (black dots in figure 34) starting from a root position and in a set direction.
- In each iteration, attraction points influence the direction in which the next branches will grow. Attraction points are the points close enough to the branches being generated. The influence is the average of the normalized vectors pointing from the previous branch's end towards the influence points. There can be one, multiple or no attraction points influencing a tree node.
- Remove attraction points that are close enough to the generated node so they don't influence the next generation of branches.

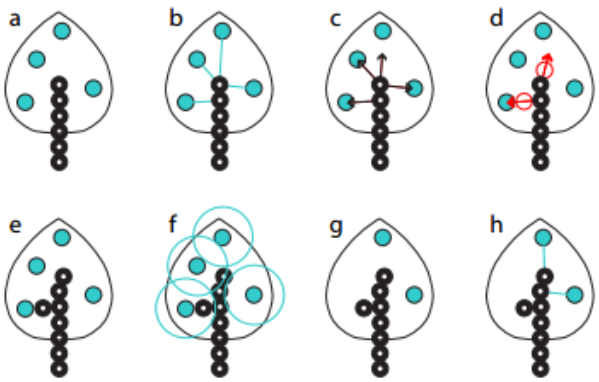

Figure 35 - Steps of the space colonization algorithm

While this technique was designed for trees generation, we noticed that road network systems have a similar growth behavior. Streets tend to grow in certain directions while being smoothly rotated by attraction factors that can lead to the creation of sub-branches, i.e. sub roads. Our first implementation of the algorithm in 3D space is shown in figure 36. We see how the branches start being generated from the root (white sphere in the top) and then start being influenced by bending or creating sub-branches as they approach attraction points (in red).

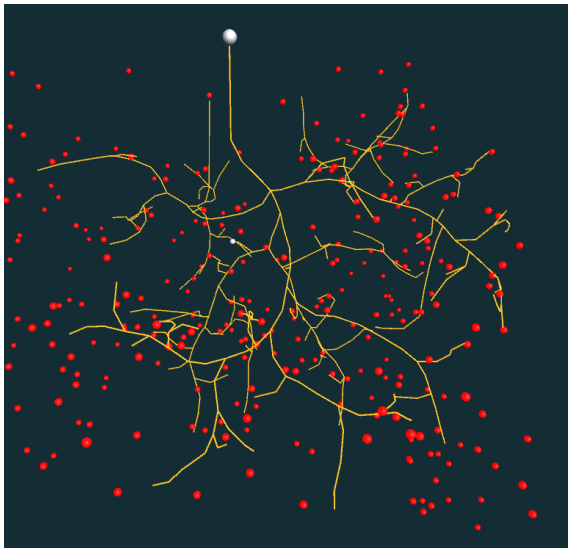

Figure 36 - First implementation of the space colonization algorithm in 3D space

We then modified the script to make it work in 2D space since it's a better approximation for how street networks are layout. We modified the random attraction points generation script to assign the points positions on the same plane. We see the result in figure 37 and we notice how the tree like shape starts to resemble an organic street network.

Figure 37 - Space colonization algorithm result in 2D space

However, the only problem with the previous example is that all the sub streets seem to be emanating from the same region and direction which may happen sometimes in real cities but isn't always the case. To remedy to this problem, we added the possibility to specify the starting position of the root node and the direction in which branches are going to grow initially. Using this modification, we can specify a root that is in a higher plane than the attraction points and start the generation process following the down direction. This gives us a more natural looking layout where streets seem to be branching from a common hub area (see figure 38).

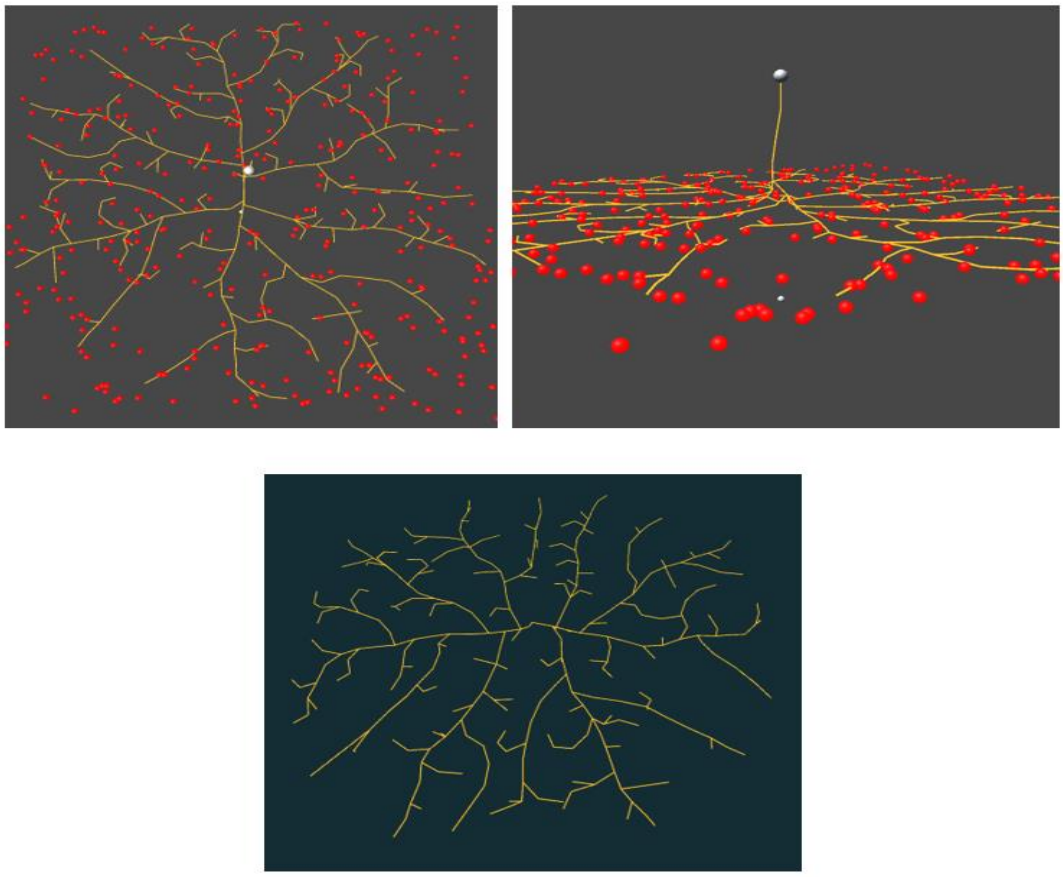

Figure 38 - Space colonization algorithm result after adding tweakable root position and initial direction

3.4 Allotment and Parcels Generation Tool

The next step in the generation process is to create lots and subdivide them to create sub parcels. These sub parcels will represent the smallest geometrical regions of the city. The latter will help us decide where to generate the various buildings, parks, parking lots, etc... We need a method that can subdivide any set of points forming a region into sub regions regardless of shape and orientation. Also the result needs to look organic and realistic. We then need a subdivision algorithm that can adapt to the shape of the whole region and give varied results.

3.4.1 Oriented Bounding Box Subdivision

The first method we tried is the Oriented Bounding Box Parceling method described in Chapter 1 section 4. Here the goal is to recursively subdivide a region by cutting the shape in half and doing intersections checks to create new sub regions. It creates roughly cuboid parcels and it works best when the initial space is also almost cubic and the expected output is similar to a Manhattan style subdivision.

The initial step of this method is to extract the convex hull of the parcel we want to subdivide. The convex hull represents the overall shape of the parcel and has the property of forming the convex perimeter that contains all the points. Several algorithms allow the creation of the convex hull from a random set of points, for our project we used the Jarvis march algorithm. In this algorithm we pick the left most point and we keep wrapping points in counterclockwise order. The challenge is given a point, how to determine the next point forming the convex hull? The solution proposed in this algorithm is to use check orientation for three consecutive points. The next point is selected if it beats all other points in counterclockwise orientation meaning that given the current point ‘p’, a point ‘q’ is accepted as the next point in the convex hull if for any other point ‘r’, orientation(p,q,r) is counterclockwise. Here is the pseudo code for the algorithm and a visual representation in figure 39.

```
// current point is p, next point is q, iteration point is r
initialPoint  = p // start with the left most point
While q != p do {
    For any point r in parcel points {
        If (Orientation (p, q, r) = counterclockwise)
            //Store that point ‘q’ is the successor of point ‘p’ in the convex hull.
            Next[p] = q;
    }
    //Change the initial point 'p’ with the next point ‘q’ just found
    p = q;
}
```

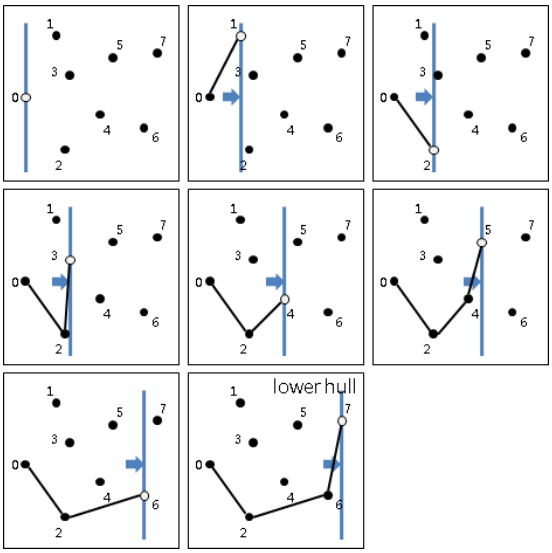

Figure 39 - Execution steps of Jarvis march algorithm for convex hull extraction

In figure 40 we can see how we extracted the convex hull from a set of random points using the Jarvis's March algorithm. As we mentioned earlier the advantage of using the convex hull is to transform the overall shape of the parcel into a convex shape that will make our future intersection calculations much more consistent.

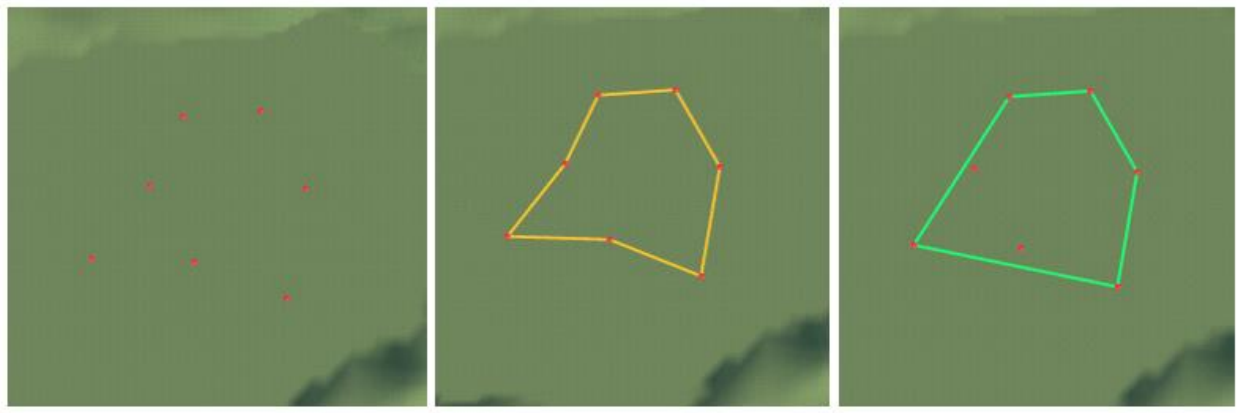

Figure 40 - (left) parcel points in red (center) parcel limits in yellow (right) convex hull of the parcel in blue

After extracting the convex hull of the parcel we're trying to subdivide, we need to find the oriented bounding box (OBB) of the shape. We will use this box to decide in which direction we're going to split the parcel to create two new sub parcels. However, a shape can have an infinite number of bounding boxes, so we decided to extract the smallest possible OBB. We know that the smallest OBB will have one of its sides parallel to one of the parcel edges. So we generated all the possible bounding boxes following this rule and we keep comparing their sizes, making sure to keep the smallest box. To generate a box along an edge we need to transform the coordinate system of the parcel points and align them onto one axis, find the bounding box, compute its size and then switch back to the original coordinate system. Here is the pseudo code for the algorithm:

```
Function FindSmallestObb( parcel ) {
        For each parcel segment {
                projectionAngle = AngleFromSegmentToXAxis(segment);
                for each parcel point
                        Rotate(point, projectionAngle);
                obb = FindOBBLimits();
                obb = AlignBoundingBox(rotatedSegment);
                For each parcel point
                        Rotate(point, -projectionAngle);
                If (obbArea < storedObbArea) {
                        storedObbArea = obbArea;
                        storedAngle = projectionAngle;
                }
        }
        finalObb = Rotate(obb, -storedAngle);
        return finalObb;
}
```

This method was particularly efficient since it's invariant to the parcel shape and orientation. We were capable of extracting the smallest fitting oriented bounding box regardless of the parcel's orientation and its convexity. Figure 41 shows the extracted OBB of the previous parcel.

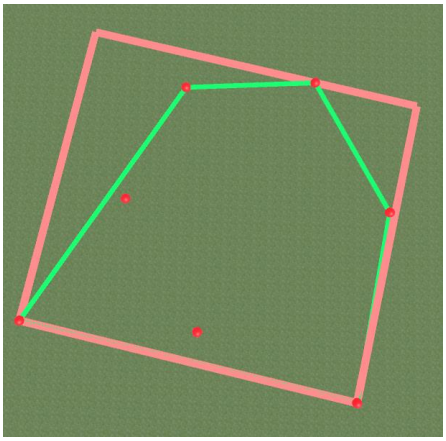

Figure 41 - Extracted OBB of a parcel in red

Now that we have the oriented bounding box, we can choose in which direction and what position we plan to subdivide the parcel. We look for the shortest side of the bounding box rectangle and we perform the cut using a parralel line. We chose the line that splits the box in half but it's also possible to use other proportions to have different results. After performing intersection checks between this cutting line and the parcel segments we are capable of finding the intersection points that we're going to use to create two new subparcels. In figure 42, we can see the subdividing line in yellow shortened between the intersection points.

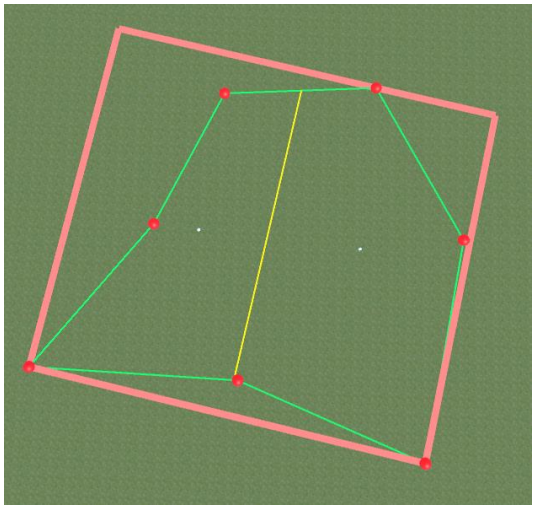

Figure 42 - Vertical cutting line (yellow) subdividing the parcel in half

After finding the intersection points, the challenge is to recreate two new subparcels from the parent parcel. In fact the intersection computation is purely based on geometrical maths in 3D space, so we don't have a the information about the order of appearance of the intersection points inside the parcel points loop. Several complex techniques exist to create polygons from a set of random abstract points; Delaunay Triangulation is probably the most popular technique for this purpose. It's commonly used in computer graphics and CAD softwares to create meshes from a set of points. However this technique is complex, uses several other techniques such as voronoi diagrams and would require a lot of time to implement properly.

We came with a solution that is more specific to our problem. The idea is to loop through the existing parcel segments and check if one of the intersections points belongs to one of these segments. Once we found the first segment that contains one of the intersections points, we create a new segment that uses the dividing point as a new end point. Then we continue looping through the rest of the segments until we find the next intersection point. We create another segment that starts from the latter dividing point and ends where the current segments ends. Now all we have to do is to close the loop by going through the rest of the points. The pseudo code for the algorithm is found below and results for different numbers of iterations can be seen in figure 43 and 44.

Figure 43 - The result of two subdivision iterations on our parcel resulting in 4 sub parcels

```
Function FindSubParcelStartingFromPoint( point ) {
        subParcel = new Parcel();
        foreach point in parcel {
                temporarySegment = new Segment(point, nextPoint);
                if ( lookingForFirstIntersectionPoint  == true ) {
                        if ( one of the intersectionPoints is in temporarySegment ) {
                                temporarySegment.endPos = firstIntersectionPoint;
                                lookingForFirstIntersectionPoint  = false;
                        }
                subParcel.addSegment(temporarySegment);
                }
                Else {
                        if ( one of the intersectionPoints is in temporarySegment ) {
                                lookingForFirstIntersectionPoint = true;
                                cuttingSegment = new Segment ( firstIntersectionPoint, secondIntersectionPoint);
                                subParcel.addSegment (temporarySegment);
                                continuitySegment = new Segment (secondIntersectionPoint, nextPoint);
                                subParcel.addSegment (continuitySegment);
                        }
                }
        }
        return subParcel;
}
```

The overall subdvision algorithm is then following these four majors steps:

- Extraction of the parcel's convex hull.
- Extraction of the oriented bounding box using the convex hull.
- Cutting the parcel in half using the oriented bounding box.
- Use the intersection points to extract two sub parcels.

We can then use this algorithm recursively to subdivide any parcel we have. The bigger the number of subdivision iterations is, the more subparcels we obtain.

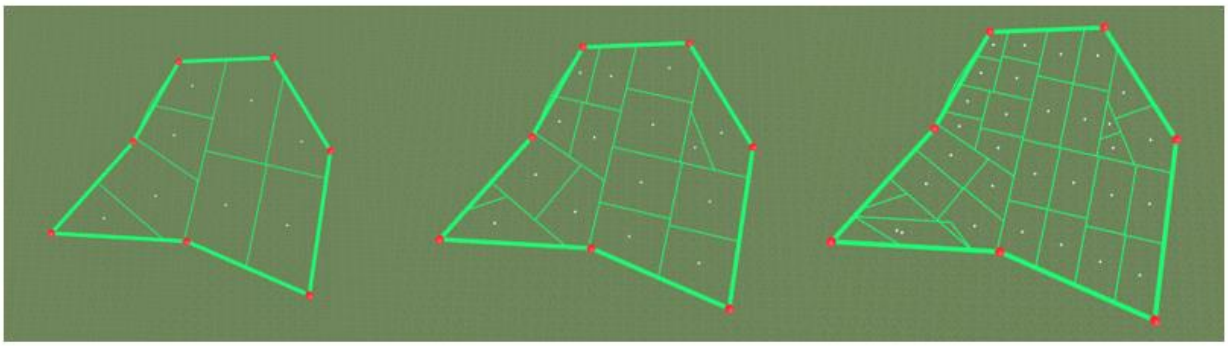

Figure 44 - OBB subdivision using respectively 2, 3 and 4 iterations

The OBB subdivision method outputs a quite convincing subdivision result. It's more suited for urban areas of modern cities like New York city, where the city layout is more cubic and follow a grid shape. This algorithm can be also further improved to only consider parcels with road access or parcel of an area within a certain size.

3.4.2 Grammar Based Subdivision

The second method we implemented for parcel subdivision is based on subdivision grammar similar to L-Systems. This method performs a series of simple subdivision tasks chosen with a random distribution to create different types of parcels and layouts.

We start with the premise that every quad can be subdivided into smaller quads and triangles, and every triangle can also be subdivided into smaller quads and triangles. We also know that any convex shape can be subdivided into quads and triangles. We then conclude that starting from any convex shape we can theoretically have an infinite number of subdivision into quads and triangles. The advantage of these two shapes is that they have very simple geometrical properties and are very easy to compute.

The goal of this method is then to create a grammar that will dictate how the subdivision process will occur. We will define a series of subdivision patterns starting from quads and triangles and then control the frequency of occurrence of each pattern. We defined the grammar we're going to use as follows:

- Q stands for quad
- T stands for triangle
- "X -> Y" is how we write our rule where X is the input and Y the output.

Using this simple grammar, it's fairly easy to create a subdivision grammar. We can say for example "Q -> QQ" which means that we're going to subdivide our quad into two smaller quads. Similarly, we can also say "T -> QQT" meaning our triangle will output a triangle and two quads. An example on how the subdivision is specified based on the grammar is shown in figure 45.

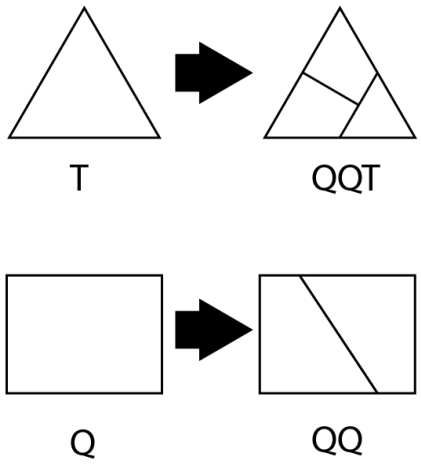

Figure 45 - Example of how grammar based subdivision works

All we have to do then is to specify different types of possible subdivisions for triangles and quads and then use a pseudo random distribution to pick which type of subdivision we're going to

use on each shape and on every iteration. We specify the occurrence probability of each type of subdivision to control the overall result of our algorithm. We can for example say that each quad Q has 45% chance to output QQ after subdivision, 30% chance to output QQT and 25% chance to output Q ("Q -> Q" means no subdivision).

Following these guidelines, this is the grammar we created for our program. We can see the resulting output in figure 46.

Rule	Occurrence chance
Q -> QQ	85%
Q -> TT	5%
Q -> QQQ	5%
Q -> Q	5%
T -> QT	90%
T -> QQT	5%
T -> T	5%

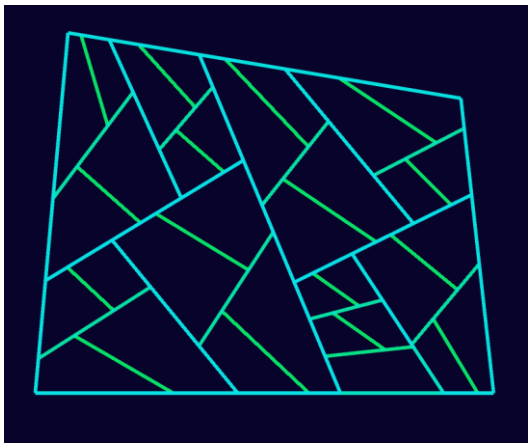

Figure 46 - Sample of a the grammar based subdivision output using 5 iterations

As we can see this subdivision method gives us a different result than the previous method. Here is the subdivision process is more random and less structured. While the OBB subdivision method outputs an almost grid style parceling, this method is more triangular and irregular. In the beginning we thought about discarding this subdivision method because the result doesn't look realistic and is messy. However, we noticed that this type of subdivision can serve a different purpose. In fact, instead of using this subdivision to extract parcels for individual buildings or monuments, we can use it to emulate secondary street networks similar to the Europeans cities. In figure 47 we have an example of Paris' street layout and we can see how it's very similar to the results we had. Also the advantage of a grammar based method is the fact that it is very easy to add more rules and define different grammars for different city layouts. We can also quickly and easily tweak the chance of occurrence of the rules and obtain various results. We can then save the grammar and use it as a template for a specific kind of secondary street layout. Also we can combine this method with the OBB subdivision method, the subdivide the triangles and quads into sub parcels.

Figure 47 - City Layout of Paris near Les Champs Elysees

3.5 Development Environment and UI Design

Our project aims to create a procedural city generation tool that can help game developers generate a variety of worlds for their games and quickly iterate in their development process. Ideally the tool would be universal and would work with any game engine, however since each engine has its own architecture it would be hard to create a generic tool that works with all of them especially under the scope of a master thesis. That's why we had to decide from the beginning on which game engine we were going to develop the tool. We quickly chose Unity3D as our main engine for this project.

This engine was a perfect solution for us for many reasons. Firstly, we have enough experience with the engine to be able to efficiently use its capabilities and extend them. Secondly, Unity3D already has a terrain system with its own API that we can use to render the generated terrain. This allows us to avoid taking care of generating the terrain mesh and implementing texturing methods, plus Unity's terrain system is already optimized to show a large number of details like grass and trees. Lastly, Unity3D seemed like a logical choice because it offers libraries to create custom editor windows that integrate perfectly in the engine's interface. It also well documented with a lot of online resources and tutorials.

As for the programming language, we used the C# language which is an object oriented language created by Microsoft. The advantages of this language is that it is derived from C++ but has the advantages of Java such as garbage collection. Using C# allows us to create a library that can be easily translated to other languages if we wanted to recreate the tool for other engines. For this reason, we used OOP paradigms and limited the use of Unity's specific methods to the bare minimum, to enable the portability of the code and the ability to add more features in the future.

The first user interface we created for our tool is the one used for the terrain generation. We created a script that extends the Custom Editor Window class of Unity. This class allows us to create a custom editor window that behaves exactly like native in-engine windows. We then have the possibility to move the window, scale it, attach it to other windows and even close it when we're not using it. Figure 48 shows some examples of layouts possible with our custom editor window. The top left image shows a floating window, top right shows how it can be attached to other Unity windows and bottom image shows how it can be embedded into other preexisting windows.

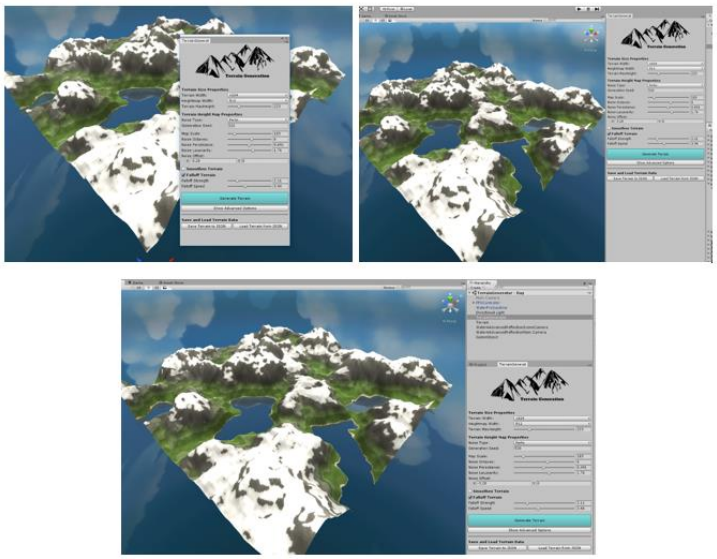

Figure 48 - Examples of different layouts possible for our custom editor window

We wanted the Terrain Generation window to provide the user with all the necessary features to be able to generate different types of terrains. The user needs to be able to control all the parameters from the same window and should be able to quickly try different settings. We also wanted the tool to be user friendly so that even a non-programmer can use it. That's why exposed

all the meaningful and useful features, making sure that it was easy to understand what every tweakable option was doing.

The first section of our terrain generation tool is about the general topography of our map. We can see the first section of the tool in figure 49. The user can choose the terrain size and the height map size from drop down menus. The user can also choose the max height of the terrain using a slider. Concerning the height map, it's generated using a noise function. So the first thing the user can do is choose what type of noise map he wants to generate: Perlin noise or Diamond Square Noise. Each type of noise shows different types of tweakable parameters. While Diamond Square noise only has a lacunarity slider, in Perlin noise we can change the noise map scale, the number of octaves, the persistence, the lacunarity and the amount of offset in X and Y axis. Of course it's also possible to type a seed number if the user wants to generate the same terrain in a future session.

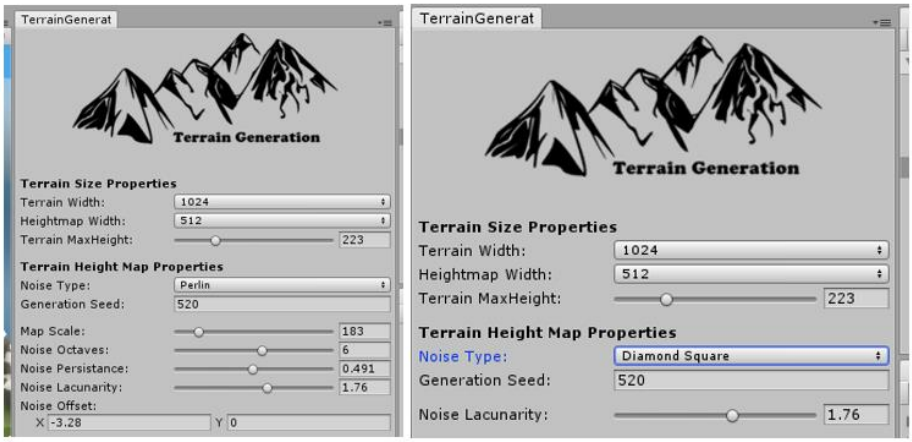

Figure 49 - Topography and height map parameters of the terrain tool

The next question is about adding more control to the overall look of the terrain. We have two main options: Smoothen Terrain and Falloff Terrain. The first option allows us to specify a

curve that we will use to interpolate the height values of the map to add more control on the terrain's shape. We can for example add a curve that will flatten the terrain and decrease the number of mountains (particularly useful in our case since we want spaces where to generate our cities). The second option allows the generation of a falloff map that we will add to our current height map to transform our terrain into an island by setting the height of the terrain's surroundings to undersea levels. Lastly in this section we have the Generate Terrain button that will apply all these parameters to the terrain and render the final result (see figure 50).

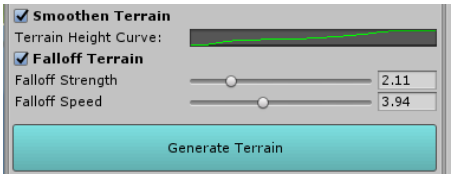

Figure 50 – The second section of the terrain generation tool is about controlling the shape of the height map

The third and last section is optional (see figure 51) and is more about adding details to the terrain. Here the user can control how the textures will appear on the terrain based on the map height. He can specify the upper limits of each texture. These limits will serve as a basis for the textures but also for other details such as grass or rocks accumulations. Of course the user can decide when to generate these details using the green buttons or clear the result using the red ones. The trees use a similar layout but instead of tweaking the upper limits, the user can control the trees' density on terrain. Finally, we added the option of saving and loading terrain presets from JSON files.

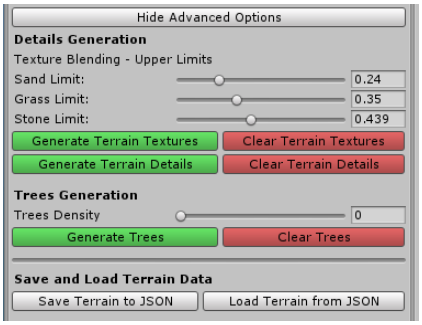

Figure 51 – The third and last section of the terrain generation tool is about generating details

For the road network generation tool and the allotment system we used a different approach for our tool UI. Instead of creating a Unity3D window like we did for the terrain, we decided to enhance the objects' inspector view by using the custom editor capabilities offered by Unity. For this purpose, we have to create scripts that extends the Editor class from the UnityEditor library. This class allows us to add buttons, UI separators, drop down menus and all kinds of UI functionalities necessary for a tool creation. Figure 52 and 53 show the difference between two Unity objects with the same script attached to them, the first one doesn't have custom editor functions while the second is extended by a custom editor.

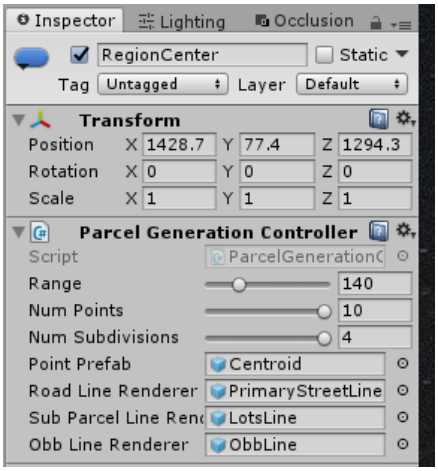

Figure 52 - Inspector window of a an object with a regular script attached to it

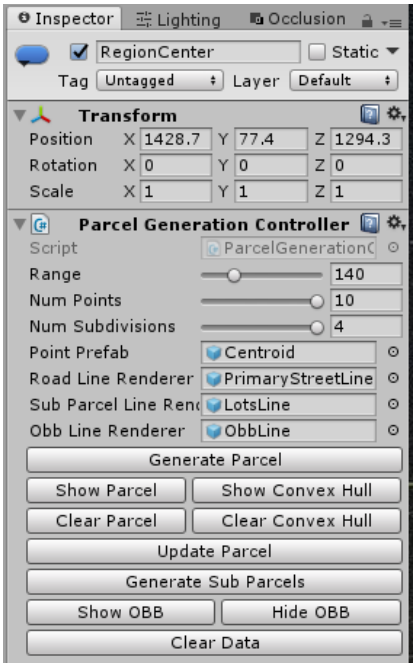

Figure 53 - Inspector window of an object with a script extended by a custom editor

One more user friendly option that we added to our tool is the use of 3d gizmos to show space information to the user while using the tool. These gizmos give a visual representation of some of the parameters the user can tweak in the tool before starting the generation process as shown in figure 54.

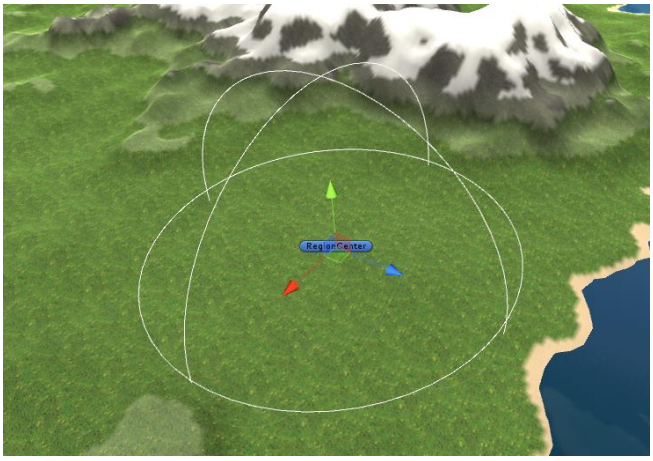

Figure 54 - Wired sphere gizmos to show a parcel's region location and area

Chapter 4. Evaluation

1. Experimental Setup

To determine if the tool met our initial goals and our design expectations, we had to run different test sessions. The purpose of these sessions is not only to gather feedback during the development process in order to fix bugs and evaluate the different functionalities of the tool but it was also a way to evaluate the output and the results of our generation algorithms. Our testing protocol was based on two different types of testing: The first is a usability test and was targeted towards the future users of the tool, i.e. game developers. The second type of testing was done using online crowdsourcing solutions where the goal was to evaluate the different outputs we could have from our tool and compare it to real life data.

4.1.1 Targeted Testing

For our usability tests we decided to perform a targeted testing phase with the help of the local WPI game developers and students. This testing session was performed on 20 game development students who volunteered to participate in this unpaid study. Since our goal was to assess the tool's functionalities and usability, we had to target an audience that was already accustomed to working with game development tools and game engines. Most of our testing sessions were performed in WPI's IMGD labs where most of the game development students gather to work on their projects. Choosing these specific locations helped us gather more testers for our tool.

Since we were testing a Unity3D tool, it wasn't possible to create external builds to be able to perform remote testing sessions. We set up some of the lab computers with the latest version of

the Unity engine and a copy of our tool. We also included a screen recording software that would record all the user's interactions with the tool for future investigation in case of bugs or to determine if some parts of the tool are unclear.

We start by briefly demoing the tool to the user making sure to explain all of its different functionalities and features. Then we would ask the user to play around with the tool for a period of 10 minutes and try to generate something they have in mind. We also explicitly ask the user to describe his intentions and speak out loud his thought process. We would take note of the most important remarks that would help us improve the tool and fix some broken features.

After the initial 10 minutes' test is over, we ask the user to try and generate a specific type of terrain. For consistency purposes we alternated between two types of terrains: either an island with large flat areas or a terrain patch with a large distribution of mountains. We decided to make the second phase last for 20 minutes to allow the tester to have time for tweaking and refining their output.

Finally, after the second phase is over we stop the recording software and we give the user a post-test survey (see Appendix D). This questionnaire will help us assess the overall usability of the tool, its usefulness and ease of use. This will not only also help determine the most successful points of our tool but also its gaps and failures.

4.1.2 Crowdsourced Testing

As we mentioned above, our second type of testing was performed using an online crowdsourcing tool. We used the website CrowdFlower a data mining and crowdsourcing website used by data scientists to create training data to build models and train machine learning

algorithms. The system will allocate a number of contributors to work on your data and will test their answers against known correct hidden answers.

This methodology was a perfect way to test different outputs from our tool on a large number of users. Our goal was to determine if the generation results of our tool can mimic real life data and fool the contributor's perception. We would consider our generation process successful if a large majority of the tester can't tell the difference between a terrain generated using our tool and a terrain generated using real life data.

To be able to compare a procedurally generated output and real life data we had to find a way to present them in a similar manner. Showing a 3D model of a terrain inside unity and comparing it to a real photography of a terrain wouldn't make much sense since the user would be able to immediately tell the difference. The solution we came up with was to present the real life data using the same visualization technique used by our tool. Since we already are able to apply a height map to the Unity terrain, all we had to do is to bring height maps of real life locations. We apply these height maps to our terrain and we texture them using the same technique we used to texture our procedurally generated terrain. Now we have a real life location rendered and represented in the same way as the output of our tool. We can see if figure 55 how we managed to render the map of Malta based on the visualization method used for our procedurally generated terrain.

Figure 55 - (left) procedurally generated terrain (right) terrain generated from the height map of Malta

Now that we found a way to evaluate the realism of our generated output, all we had to do is to gather a set of screenshots taken using the technique described above and pass it through a test. We wanted to make the test as simple as possible to avoid confusion during the test. Since this time the testers are selected by the system, nothing ensures that they are game developers, it's actually very unlikely that they are. So we had to make sure that the instructions clear and simple.

We selected 20 screenshots for our test, 10 are generated using real life data and 10 are procedurally generated by our tool. Each tester will individually examine each image from the set and answer a simple question: "Is this terrain generated from real data or is it computer generated?". We also made sure to include an instructions page to give more insights on the test protocol and its steps, including a visual example at the bottom. We can see the instructions page in figure 56. We performed two rounds of crowdsourced testing. The first round included 24 contributors paid approximately $0.12 each ($2.88 total). In the second round we doubled the size of the testers to 50, paid each approximately $0.1224 ($6.12 total).

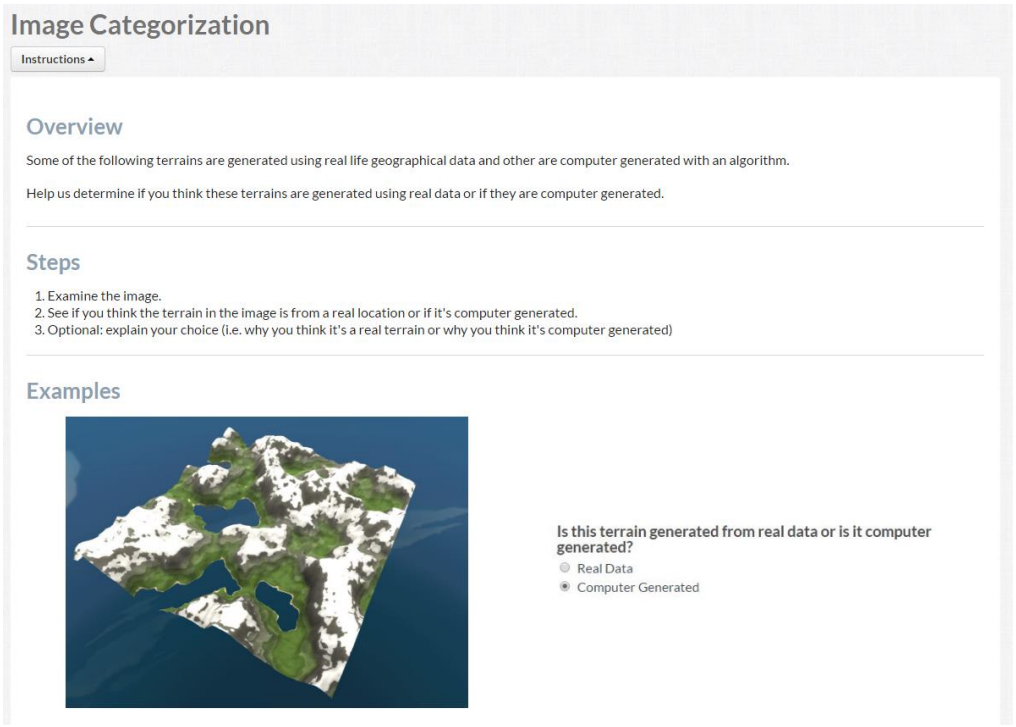

Figure 56 - Instructions page used for the crowdsourcing test

4.2 Hypothesis

As we said in the section above, the targeted testing was more used for iteration purposes and for bug fixing. We wanted to assess the tool's features and functionalities during the development process. However, using the post survey test we are able to extract some general trends about user satisfaction, the tool's usability and future use.

For the results focused on these four categories that we extracted from our survey:

- General PCG questions
- Tool usability
- Feature Assessment
- Potential Use

Most of these categories give us insight on the development process of the tool and helps us consolidate our successes and also notice our failure. The results we expect the most are from the "Potential Use" category. We are expecting that most of users were interested by the tool and see themselves using it for their game development projects.

Concerning the crowdsourced testing, the expected results are more straightforward. The goal of the test was to see if people can tell the difference between the terrains generated with our tool and terrains that were created using real height maps. A high success rate in the test can tell us that our terrains don't really look realistic and people can easily notice them. However, a low success rate can lead us to think the test isn't well constructed and testers were either confused or didn't understand the goal of the test. For this reason, we are expecting an average level of success in the test. Approaching the 50% success rate would suggest that the testers could really not tell the difference between the two types of terrains.

4.3 Results

4.3.1 Targeted Testing Results

- General PCG Questions

The first two questions of our survey were about the familiarity of the testers with PCG. We asked the tester if they used PCG tools in their game development projects in the past. As we can see in figure 57, the majority of the tester gave negative answers or were uncertain. Only 25% claimed they used PCG tools before.

Figure 57 - Percentage of testers that used PCG before

We also wanted to know the testers point of view on PCG tools as a way to fasten development. In the chart presented in figure 58 we see how 80% of the testers confirm that statement.

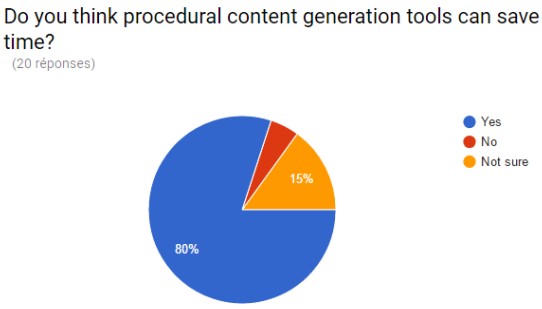

Figure 58 - Percentage of users that think PCG can save development time

- Tool usability

The next of questions was about the tool usability and clearness. We asked the tester if the tool seemed confusing. The results presented in figure 59 show that half of the users said that it was sometimes confusing to use the tool and 25% were either confused or unsure. Despite being a negative result, it was really helpful to have this feedback since we used it as a motivation to create documentation for the tool.

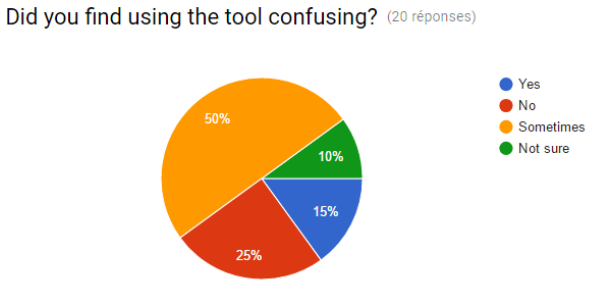

Figure 59 - Percentage of users that thought the tool was confusing

Following the previous question, we made sure to add a question to locate the most confusing features of the tool. In figure 60 we see how 18 out of the 20 testers (94.7%) thought the road network part of the tool was the most confusing feature, followed by the allotment and parcel subdivision. We were expecting these results, since these features are incomplete and still in a development state.

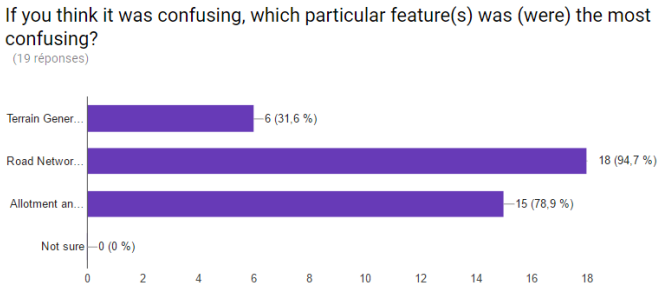

Figure 60 - The most confusing features in the tool according to the testers

When asked if they think the tool needs more documentation, 45% answered positively (see Figure 61). This is again another proof that tool needs documentation, especially considering its complex nature.

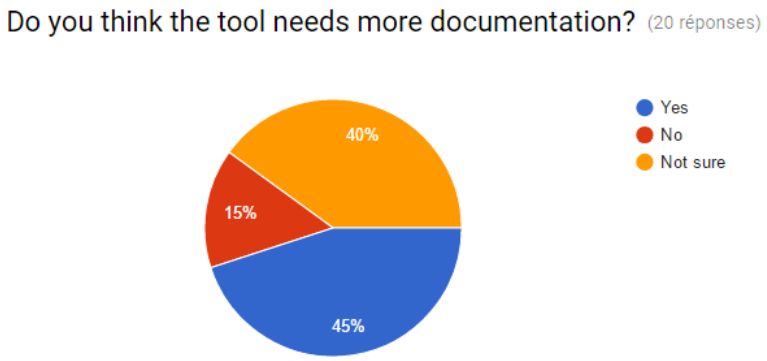

Figure 61 - Percentages of users that think the tool needs more documentation

Finally, we asked the tester what kind of documentation they would prefer. Most of the users said that instruction videos would be the best form of documentation, followed by written manual chosen by 66.7% of the users (see Figure 62).

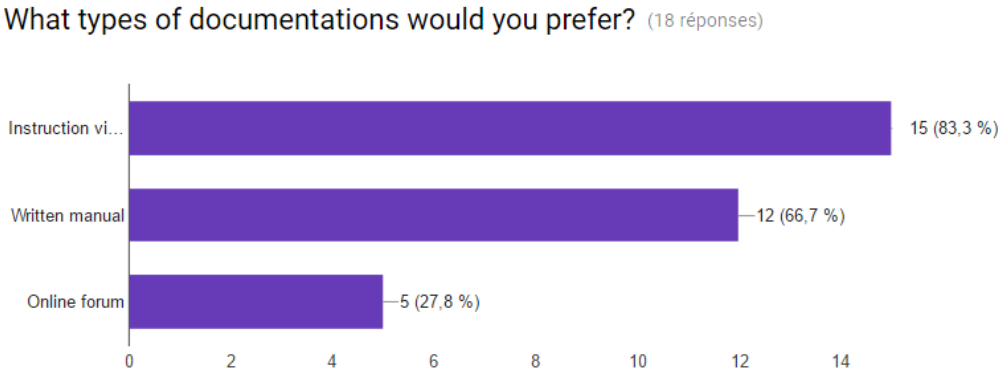

Figure 62 - Preferred types of documentation according to the testers

- Feature Assessment

Next we wanted to ass the three main feature of tool: Terrain Generation, Road Network Generation and Allotment/Parcel Subdivision.

When asked about which particular features the users found interesting (figure 63), the results were quite promising since more than 80% of the testers found the terrain generation and allotment subdivision interesting. Even the road network generation part had an above average result with 65% despite being still in an early development phase.

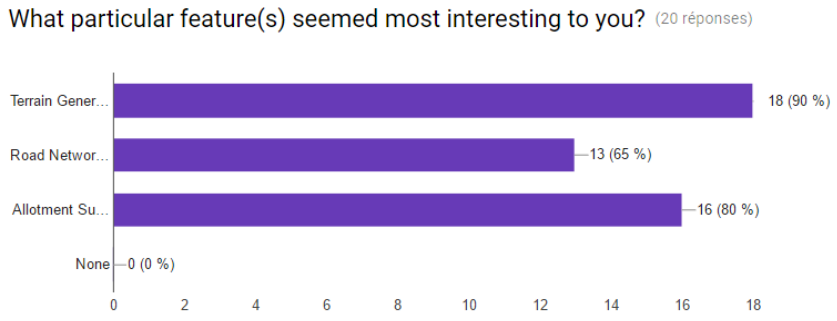

Figure 63 - Percentages of interest for each feature of tool

In figure 64 we have another result that was kind of expected. The terrain generation part of the tool is the most complete feature, and with a 90% approval rate we can decide with confidence to assign less development time for that particular feature in future and focus more on the rest.

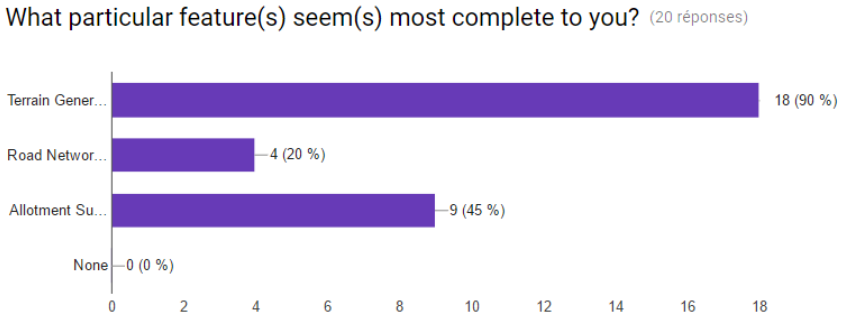

Figure 64 - Most complete features of the tools according to the users

A contrasted but also expected result was also found when we asked about which feature they think needs more work. As we can in the chart in figure 65, a vast majority think the road network needs more work, followed by the allotment subdivision. Also 45% thought the UI could also need more work, which we would take into consideration for our future development plans.

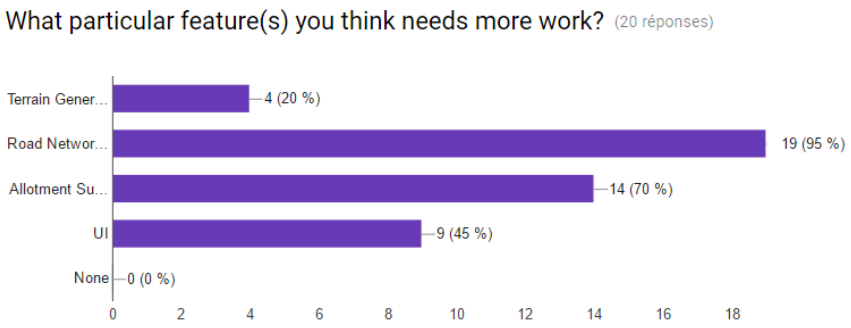

Figure 65 - Features that need more work according to the testers

- Potential Use

Finally, we asked the users about their potential use of *Urbis Terram* in its current state and once its development is complete. In figure 66 we see how 85% of the users potentially might want to use the tool in its current state and half of these testers are willing to start using it now.

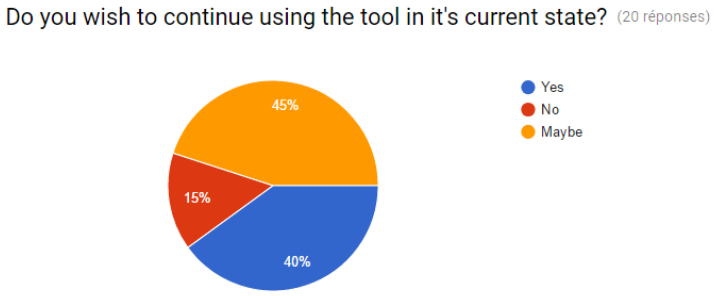

Figure 66 - Potential use of the tool in its current state

Even more promising results are found when asked if they wanted to continue using the tool once it's complete (see figure 67). Here 75% answered positively to the question and reflects the potential the final tool can have.

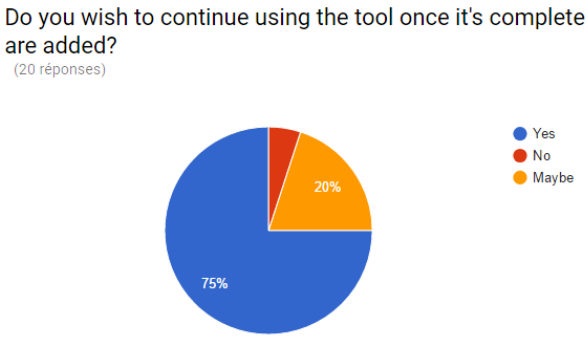

Figure 67 - Potential use of the tool in its final state

4.3.2 Crowdsourced Testing Results

As mentioned above, we performed two rounds of crowdsourced testing. The first round had 25 contributors and the second 50. Here the results are more straightforward and are quite easy to process since we're only assessing the degree of which testers can tell the difference between our tool's terrains and terrains generated using real height maps.

Each tester will have to answer all 20 questions, specifying each time the type of terrain he thinks he's seeing. While preparing the test, we set up the parameters to consider the test as failed if the tester makes 50% or more wrong answers. In fact, we're not testing how good the users are performing in our test, but in the contrary we're testing how badly they're performing.

In the first testing round, only 8 out of 24 contributors (33%) succeeded the test while 16 failed. This means that 67% of the testers failed the test by giving more than 50% wrong answers. We can see the results of the first round in figure 68.

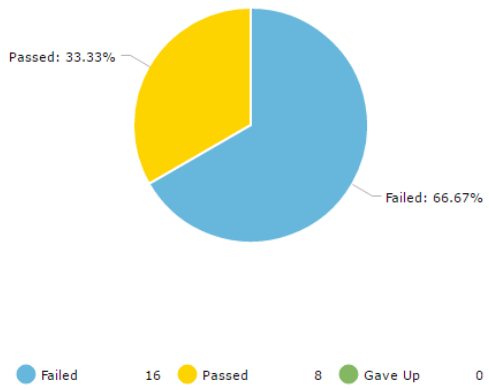

Figure 68 - First crowdsourced testing results (25 testers)

For consistency reasons we performed a second round of testing while doubling the number of contributors this time. From the 50 testers, only 15 answered correctly (30%), 34 gave wrong answers (68%) and 1 gave up the test. A visual representation of the results is shown in figure 69.

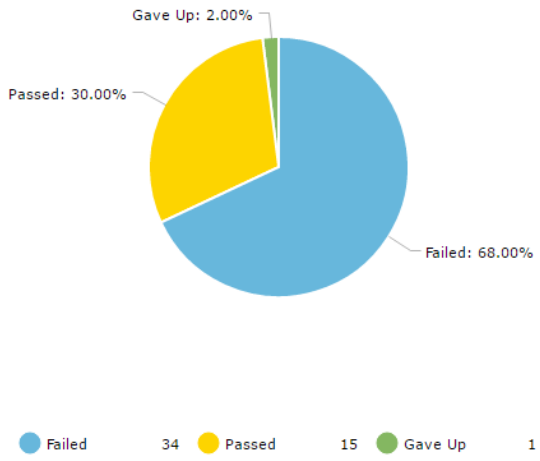

Figure 69 - Second crowdsourced testing results (50 testers)

The two rounds of test gave almost the exact same result which confirms the consistency of the test. We aggregated both results into one single chart shown in figure 70.

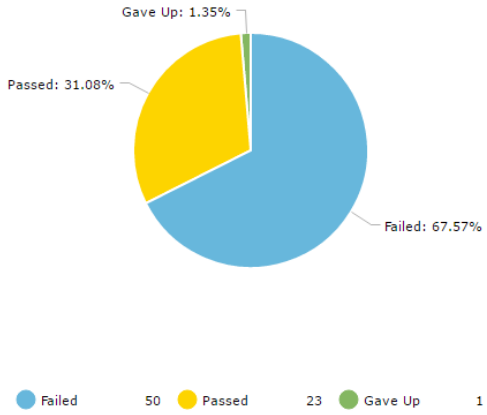

Figure 70 - Aggregated test results for both rounds (74 testers)

While we can consider this result as a primary success since two thirds of the testers failed the test, these results alone are not sufficient to be able to conclude on the success of the generation process. In fact, we know that a majority of testers failed the test but we also need to investigate the primary reason of their failures. On one hand, if most of the wrong answers they gave came from questions about the terrains generated from real height maps, it doesn't give us insight on the level of realism of our generated terrains and also points to the fact that the test was poorly designed. On the other hand, if most of the wrong answers are about the terrains generated using our tool, we can conclude that our output was convincing enough to fool the testers.

The ideal case would be then if the user fails equally on both types of terrains. That would suggest that the test was well designed and that both types of terrains are similar enough to create confusion. For this reason, we looked at individual results for each terrain image. We looked separately at the results for the users who failed the test and also for the users that succeeded.

Looking at the charts in figure 71 and 72, we see how the testers that passed the test had wrong guesses on average for three real terrains (real terrain 3, 4 and 6) and four PCG terrains

(PCG terrain 1, 6, 8 and 9). While we can't consider the testers who passed the test to conclude on the efficiency of the tool to output realistic terrains, we can see once again that the test was well designed since on average the testers failed almost equally on both types of terrains.

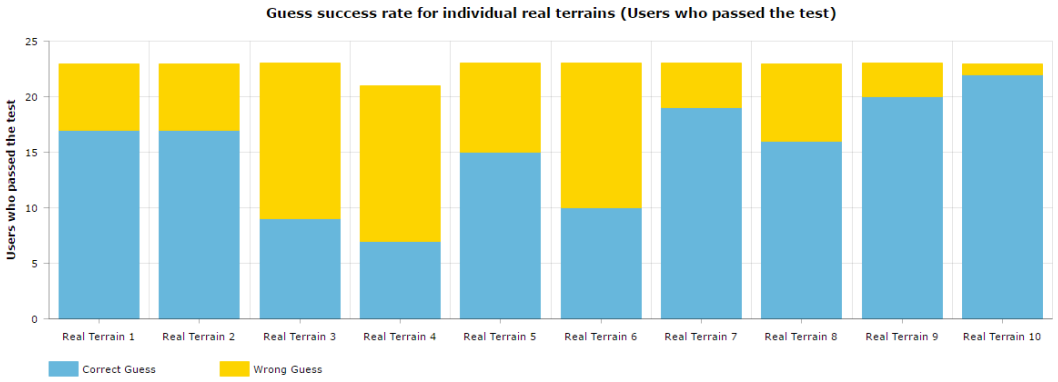

Figure 71 - Guess success rate for each real terrain image (results for 23 testers who passed the test)

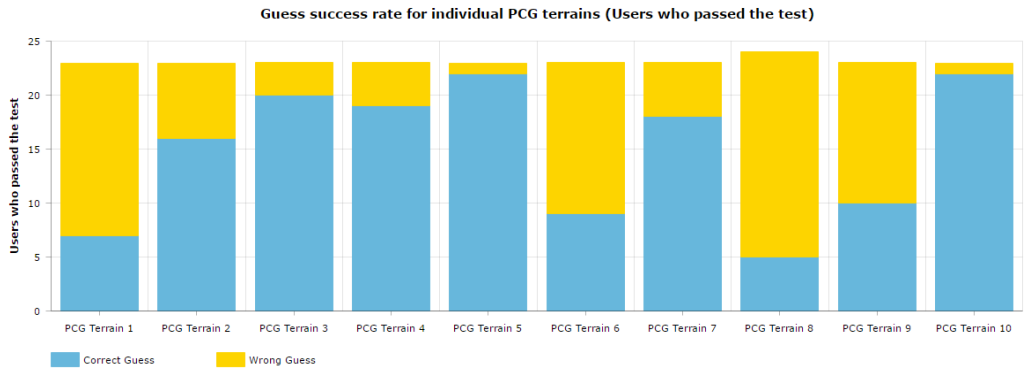

Figure 72 - Guess success rate for each PCG terrain image (from 23 testers who passed the test)

The results that interest us the most are actually from the users that failed the test. As mentioned above these users (who represent two thirds of the total number of testers) had more than 50% wrong answers on the test and thus are good candidates to check if the source of their error was the real or the PCG terrains. In figure 73, we can see how on average the testers managed

to guess wrongly 5 real terrains (real terrain 3, 4, 5, 6 and 8). This is already a good result since it suggests that it's not solely because of miss guessing real terrains that these users failed the test. Since 5 images represents 25% of the total number of terrains, the user must have wrongly guessed at least 50% of the PCG terrains too. And this is confirmed by the results shown in figure 74 for the users who failed the test. Only 3 PCG terrains were on average detected (PCG terrain 2, 5 and 10), meaning that on average 70% of the PCG terrains fooled the testers. These results support our hypothesis and are considered as a success since it provides evidence that the tool can produce realistic looking terrains in its current state, however there is still room for improvement to increase even more the degree of realism of the results.

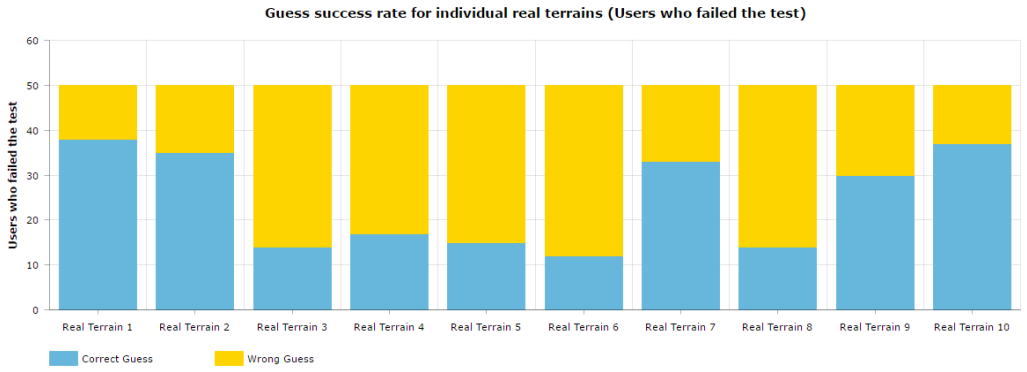

Figure 73 - Guess success rate for each real terrain image (results for 50 testers who passed the test)

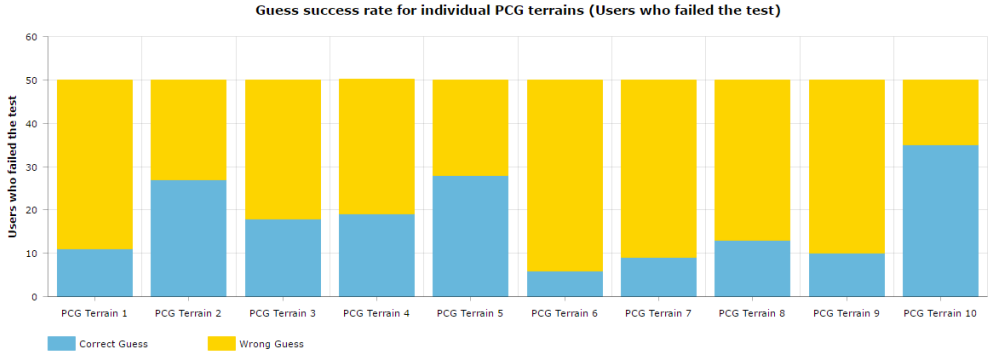

Figure 74 - Guess success rate for each PCG terrain image (results for 50 testers who passed the test)

Chapter 5. Conclusion

In this section we discuss the successes and failures of our project and the reasons behind them. We are also going to state our future plans for the tool taking the testing results in consideration.

5.1 What went right?

First of all, the biggest motivation of this project was to tackle the technical challenge of creating a procedural city generation tool for Unity3D. And in that regard, this project was really successful. As the project was too ambitious, we had the chance to work on several complex problems and had to come up with ways to overcome difficult geometrical computations. Our technical knowledge has expanded drastically while working on this project, especially in terms of managing data structures, creating a scalable code but also on how to create tools and custom UIs in Unity3D.

Secondly, the terrain generation tool was a really big success. It was the most complete feature and produced a lot different types of terrains. Most of the testers were asking about the release date of the tool, especially level designers. They thought that the terrain generation tool will help them speed up their development process and wanted to start using it in its current state. Also as our testing results show, most of the contributors couldn't tell the difference between real terrains and our own terrains, which is again considered as a major achievement.

Finally, while the allotment subdivision system was extremely complicated to achieve we managed to create a robust framework that is also platform independent since it's mainly based on geometrical computations. The two subdivision methods we implemented gave amazing results

for different types of shapes and even non convex shapes. This framework will be really helpful for the future of the project.

5.2 What went wrong?

The biggest problem we encountered during our development process, especially by the end of the project is without surprise scoping. The initial scope of the project was too ambitious and didn't take into account the complexity of the project. In our proposal we thought that we were going to be able to finish the whole tool by the end of the first semester and spend the second semester working on a game that uses its functionalities. However, this turned out to be completely unrealistic especially that this is a one-member team. In an article published on 80 level called procedural world building in Ghost Recon Wildlands [15], Ubisoft employees explain how they created a PCG tool to generate their game world. They stated that road generation and buildings placement was an extremely complex problem to solve. To generate their terrain with cities on top of it, it takes them a whole computer farm working for over 20 minutes to achieve their result whenever they make any change in the terrain's topology.

Following the previous point, time was also a big constraint in our development process especially in the second semester. We had to travel for several weeks to attend different conferences and that took away from development time that We could have used to polish some of the features and for bug fixing.

Finally, we faced a lot of scalability issues and that slowed us down by the end of the development phase. The problem was that the data structure used for the terrain, the roads and parcel subdivision were independent in the beginning since we worked on these problems separately. When we decided to merge all three features in one single tool, we noticed that some

of the data structures were either redundant or incompatible thus preventing us from having a fully automatic generation process.

5.3 Future work

As our testing results show, most of the testers stated that they would use the tool once it's complete and when more features are added. This motivates us to continue working on the tool, finishing the remainder of its features, making sure to deliver a quality product in the end.

We are planning on revisiting all the data structures and make sure that the road network generation system can take into account more accurately the terrain's elevation. Also a lot of the code that was written during the early stages of the project should be revisited and modified to facilitate the addition of features in the future.

In addition to that, right now we have the possibility to save the terrains we generated into Json files and it would be a great addition to have templates of certain types terrains that the users can use as a starting point in their generation process. For example, we can add templates for steppes, islands, mountainous areas, tundra, canyons… and other types of terrains and store those in files. The user can then load any specific type of terrains he wants by locating the specific Json file.

Finally, we are planning to allocate a large amount of time to polish and bug fixing. Our main focus would be to guarantee robust and optimized generation algorithms making the tool reliable and professional. We are also going to devote important efforts into improving the overall user experience and user interface of the tool. The generation process should be as seamless and intuitive as possible to the user. We will also make an extensive documentation to help the users understand what each specific feature of Urbis Terram. A written documentation would explain

all the required steps to generate realistic looking cities and will also explain the purpose of each individual parameter. Tutorial videos were also largely requested by our testers and would help them have a more visual and interactive way to learn how to use the tool. By the end of this phase we plan to publish *Urbis Terram* as a standalone unity package on the Unity asset store.

5.4 Conclusion

The goal of the project was to create an innovative tool that would help game developers and more specifically level designers generate realistic looking cities for their games. The tool would give them a good starting point for their project, saving them a lot of tedious manual labor and also precious development time. They would have the possibility to manually modify the result in a later phase to make it more fit their needs.

By the end of this master's project, we managed to partially approach our initial goal. The initial scope turned out to be too ambitious for a one-year/one-person project. However, despite these difficulties we managed to create an advanced prototype for our tool's features. The terrain generation tool was particularly successful with testers and provided realistic looking results. And with parts of the other features being already implemented, we plan to continue working on this project to achieve its initial goals and create a really strong and robust tool for cities generation.

In terms of personal and skill development, this project was also really successful. The development was extremely challenging since the beginning and was a great opportunity to learn how to implement procedural content generation algorithms and also how to create custom UI inside the Unity editor. It was also an amazing exercise in time management, gaining more experience on how to scope bigger projects and how to set smaller milestones along the way during the development process to help keep control of the project's flow.

Finally, we think this tool is really promising and can have a lot of potential for the future. Its innovative idea of combining different types of procedural content generation processes into one single tool was well received by the testers that showed a large interest in the tool in its current state and especially in its future final state. Also, the lack of direct competitors on the asset store could help *Urbis Terram* stand out and become the new standard in procedural city creation.

References

[1] Togelius, Julian, et al. "What is procedural content generation? Mario on the borderline." Proceedings of the 2nd International Workshop on Procedural Content Generation in Games. ACM, 2011.

[2] Tom Hatfield. "Rise of The Roguelikes: A Genre Evolves." GameSpy. Jan 29, 2013 Web. 24 Sept. 2016.

[3] "Procedural Content Generation Wiki." Minecraft. Web. 24 Sept. 2016.

[4] Richard Moss. "7 Uses of Procedural Generation That All Developers Should Study." Gamasutra Article. January 1, 2016. Web. 24 Sept. 2016.

[5] Cook, Adam. "6 Mind-blowing No Man's Sky Numbers." No Man's Sky Video Game Numbers and Statistics. 09 May 2016. Web. 24 Sept. 2016.

[6] Olsen, Jacob. “Realtime Procedural Terrain Generation”. University of Southern Denmark. Oct 31, 2004.

[7] Génevaux, Jean-David, et al. "Terrain generation using procedural models based on hydrology." *ACM Transactions on Graphics (TOG)* 32.4 (2013): 143.

[8] Archer, Travis. "Procedurally generating terrain." 44th annual midwest instruction and computing symposium, Duluth. 2011.

[9] Kelly, George, and Hugh McCabe. "Citygen: An interactive system for procedural city generation." Fifth International Conference on Game Design and Technology. 2007.

[10] Vanegas, Carlos A., et al. "Procedural generation of parcels in urban modeling." *Computer graphics forum*. Vol. 31. No. 2pt3. Blackwell Publishing Ltd, 2012.

[11] Parish, Yoav IH, and Pascal Müller. "Procedural modeling of cities." Proceedings of the 28th annual conference on Computer graphics and interactive techniques. ACM, 2001.

[12] Groenewegen, Saskia, et al. "Procedural City Layout Generation Based on Urban Land Use Models." Eurographics (Short Papers). 2009.

[14] Saudergh J. "Procedural City Generation in Python - Documentation." Procedural City Generation in Python. Web. 24 Sept. 2016.

[15] Erwin Heyms. "Procedural World Building in Ghost Recon Wildlands." 80 level. April 20, 2017. Web. 20 Apr. 2017.

Appendix

A. GenerateNoiseMap (PerlinNoise Class)

```
public static float[,] GenerateNoiseMap(int mapWidth, int mapHeight, int seed, float scale, int octaves,
                                                    float persistance, float lacunarity, Vector2 offset)
{
        float[,] noiseMap = new float[mapWidth, mapHeight];

        System.Random prng = new System.Random(seed);
        Vector2[] octaveOffsets = new Vector2[octaves];

        for (int i = 0; i < octaves; i++)
        {
            float offsetX = prng.Next(-10000, 10000) + offset.x;
            float offsetY = prng.Next(-10000, 10000) + offset.y;

            octaveOffsets[i] = new Vector2(offsetX, offsetY);
        }

        if(scale <= 0)
        {
            scale = 0.001f;
        }

        float maxNoiseHeight = float.MinValue;
        float minNoiseHeight = float.MaxValue;

        float halfWidth = mapWidth / 2f;
        float halfHeight = mapHeight / 2f;

        for (int y = 0; y < mapHeight; y++)
        {
            for (int x = 0; x < mapWidth; x++)
            {
                float amplitude = 1;
                float frequency = 1;
                float noiseHeight = 0;

                for (int i = 0; i < octaves; i++)
                {
                    float sampleX = (x - halfWidth) / scale * frequency + octaveOffsets[i].x;
                    float sampleY = (y - halfHeight) / scale * frequency + octaveOffsets[i].y;

                    float perlinValue = Mathf.PerlinNoise(sampleX, sampleY) * 2 - 1;
                    noiseHeight += perlinValue * amplitude;

                    amplitude *= persistance;
                    frequency *= lacunarity;
                }

                if (noiseHeight > maxNoiseHeight)
                {
                    maxNoiseHeight = noiseHeight;
                }
                else if (noiseHeight < minNoiseHeight)
                {
                    minNoiseHeight = noiseHeight;
                }

                noiseMap[x, y] = noiseHeight;
            }
        }
        for (int y = 0; y < mapHeight; y++)
        {
            for (int x = 0; x < mapWidth; x++)
            {
                noiseMap[x, y] = Mathf.InverseLerp(minNoiseHeight, maxNoiseHeight, noiseMap[x, y]);
            }
        }

                return noiseMap;
    }
```

B. GenerateNoiseMap (DiamondSquare Class)

```
public static float[,] GenerateNoiseMap(int size, int seed, float rigosity)
{
        Random.InitState(seed);
        float[,] heightMap = new float[size, size];

        float scale;
        rigosity /= 1000f;
        //float rigosity = 0.0005f;

        // init
        heightMap[0, 0] = Random.Range(0f, 1f);
        heightMap[0, size - 1] = Random.Range(0f, 1f);
        heightMap[size - 1, size - 1] = Random.Range(0f, 1f);
        heightMap[size - 1, 0] = Random.Range(0f, 1f);


        int i = size - 1;
        while (i > 1)
        {
            int id = i / 2;
            scale = i * rigosity;

            for (int x = id; x < size; x += i)
            {
                for (int y = id; y < size; y += i)
                {
                    float average = (heightMap[x - id, y - id] + heightMap[x - id, y + id] +
                                     heightMap[x + id, y + id] + heightMap[x + id, y - id]) / 4f;
                    heightMap[x, y] = average + Random.Range(0f, 1f) * 2 * scale - scale;
                }
            }

            bool isPair = true;
            for (int y = 0; y < size; y += id)
            {
                for (int x = (isPair) ? id : 0; x < size; x = (i % 2 == 0) ? x + i : x + i - 1)
                {
                    float sum = 0;
                    int n = 0;
                    if (x >= id)
                    {
                        sum = sum + heightMap[x - id, y];
                        n++;
                    }
                    if ((x + id) < size)
                    {
                        sum += heightMap[x + id, y];
                        n++;
                    }
                    if (y >= id)
                    {
                        sum += heightMap[x, y - id];
                        n++;
                    }
                    if ((y + id) < size)
                    {
                        sum += heightMap[x, y + id];
                        n++;
                    }
                    sum /= n;
                    heightMap[x, y] = sum + Random.Range(0f, 1f) * 2 * scale - scale;
                }
                isPair = !isPair;
            }

            i = id;
        }
        return heightMap;
}
```

C. FindSubParcelStartingFromPoint

```
// Find sub parcel starting from extrema point (top most, bot most, right most, left most)
    private Parcel FindSubParcelStartingFromPoint(Vector3 point, int subParcelNumber, int iter)
    {
        // Generate Sub Parcel
        Parcel subParcel = new Parcel(parcel.name + "-" + subParcelNumber.ToString(), iter);

        bool pointFound = false;
        int pointIndex = 0;

        // Find extrema point in parcel points and keep track of its index
        // We will use that index to start iterating through the parcel points
        for (int i = 0; i < parcel.parcelPoints.Count; i++)
        {
            if (Vector3.Distance(point, parcel.parcelPoints[i]) <= 0.1f)
            {
                pointFound = true;
                pointIndex = i;
            }
        }

        // Counter for parcel points iteration
        int counter = 0;

        Vector3 firstIntersectionPoint = Vector3.zero;
        bool lookingForSecondIntersectionPoint = false;

        if (pointFound)
        {
            while (counter < parcel.parcelPoints.Count)
            {
                // The current point and the point after it form the segment for the intersection check
                int nextPointIndex = (pointIndex + 1) % parcel.parcelPoints.Count;

                Vector3 nextPoint = parcel.parcelPoints[nextPointIndex];

                // Temporary segment for intersection check
                Segment tempSeg;
                if (counter == 0)
                {
                    // First segment created, we use the point of reference (extrema)
                    tempSeg = new Segment(point, nextPoint);
                }
                else
                {
                    tempSeg = new Segment(parcel.parcelPoints[pointIndex], nextPoint);
                }

                if (!lookingForSecondIntersectionPoint)
                {
                    for (int i = 0; i < intersectionPoints.Count; i++)
                    {
                        // If intersection point is included in current temporary segment
                        if (tempSeg.PointInSegment(intersectionPoints[i], tempSeg))
                        {
                            // The end position for the temporary segment becomes the intersection point
                            tempSeg.finalPos = intersectionPoints[i];

                            // Store the intersection point
                            // To be used with next intersection point to create the new segment for sub parcel
                            firstIntersectionPoint = intersectionPoints[i];
                            lookingForSecondIntersectionPoint = true;

                            break;
                        }
                    }

                    subParcel.parcelSegments.Add(tempSeg);
                }
                else
                {
                    for (int i = 0; i < intersectionPoints.Count; i++)
                    {
                        if (tempSeg.PointInSegment(intersectionPoints[i], tempSeg))
                        {
                            lookingForSecondIntersectionPoint = false;

                            // Add the segment between the two intersection points
                            Segment intersectSegment = new Segment(firstIntersectionPoint, intersectionPoints[i]);
                            subParcel.parcelSegments.Add(intersectSegment);

                            // Add the continuity segment starting from the second intersection point
                            Segment continuitySegment = new Segment(intersectionPoints[i], nextPoint);
                            subParcel.parcelSegments.Add(continuitySegment);
                        }
                    }
                }

                pointIndex = (pointIndex + 1) % parcel.parcelPoints.Count;
                counter++;
            }
        }
        else
        {
            Debug.Log("Point: " + point + " Not Found in " + parcel.name + "-" + subParcelNumber.ToString());
        }

        subParcel.UpdateParcelPointsFromSegments();
        return subParcel;
    }
```

D. Post-Test Survey

Urbis Terram - User Testing

Thank you for testing our procedural city generation tool for Unity3D.
Please help us improve the tool by answering the following questionnaire.

How much did the tool hold your interest?

	1	2	3	4	5	
Low	○	○	○	○	○	High

Did you use PCG tools for your game development projects in the past?

- ○ Yes
- ○ No
- ○ Not sure

Do you think procedural content generation tools can save you development time?

- ○ Yes
- ○ No
- ○ Not sure

How difficult was the tool to use?

	1	2	3	4	5	
Very Easy	○	○	○	○	○	Very Difficult

Were you able to recreate the instructed result?

- ○ Yes
- ○ No
- ○ Not sure

Do you think you need more control on the output of the tool?

- ○ Yes
- ○ No
- ○ Not sure

Did you find using the tool confusing?

- ○ Yes
- ○ No
- ○ Sometimes
- ○ Not sure

If you think it was confusing, which particular feature(s) was (were) the most confusing?

- ☐ Terrain Generation Tool
- ☐ Road Network Generation Tool
- ☐ Allotment and Subdivision Tool
- ☐ Not sure

Do you think the tool needs more documentation?

- ○ Yes
- ○ No
- ○ Not sure

What types of documentations would you prefer?

- [ ] Instruction video
- [ ] Written manual
- [ ] Online forum

Do you think the tool's UI needs improvement?

- ○ Yes
- ○ Maybe
- ○ Not Really
- ○ No
- ○ Not sure

What particular feature(s) seemed most interesting to you?

- [ ] Terrain Generation Tool
- [ ] Road Network Generation Tool
- [ ] Allotment Subdivision Tool
- [ ] None

What particular feature(s) seem(s) most complete to you?

- [ ] Terrain Generation Tool
- [ ] Road Network Generation Tool
- [ ] Allotment Subdivision Tool
- [ ] None

What particular feature(s) you think needs more work?

- [ ] Terrain Generation Tool
- [ ] Road Network Generation Tool
- [ ] Allotment Subdivision Tool
- [ ] UI
- [ ] None

Do you have any feature in mind that can be added to the tool?

Votre réponse

How accustomed are you to playing city building games?

	1	2	3	4	5	
Low	○	○	○	○	○	High

For what types of games you see this tool being used for?

- [ ] RTS
- [ ] FPS/TPS
- [ ] Racing Games
- [ ] City Builders
- [ ] RPG
- [ ] Sandbox
- [ ] Platformer
- [ ] Survival
- [ ] Horror / Stealth

Do you wish to continue using the tool in it's current state?

- ◯ Yes
- ◯ No
- ◯ Maybe

Do you wish to continue using the tool once it's complete and more features are added?

- ◯ Yes
- ◯ No
- ◯ Maybe