Bachelor of Science in Computer Science
June 2017

Procedural city generation using Perlin noise

Niclas Olsson
Elias Frank

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science. The thesis is equivalent to 10 weeks of full time studies.

Contact Information:
Author(s):
Niclas Olsson
Elias Frank
E-mail:
niod12@student.bth.se
elfa14@student.bth.se

University advisor:
Hans Tap, PhD
Department of Creative Technologies

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Context. Procedural content generation is to algorithmically generate content. This has been used in games and is an important tool to create games with large amounts of content using fewer resources. This may allow small developers to create big worlds, which makes the investigation into this area interesting.
Objectives. The Procedural generation of cities using Perlin noise is explored. The goal is to find out if a procedurally generated city using Perlin noise is viable to use in games.
Methods. An implementation generating cities using Perlin noise has been created and a user study along with data collection tests the cities' viability in games.
Results. The implementation succeeds with all the technical requirements such as performance and determinism. The user study shows that the cities created are perceived as viable in games.
Conclusions. The cities generated with the implementation seems to be viable in games. The results show that the generated content are percieved as more viable than random generated cities. Furthermore the generation speed is fast enough to be used in an online setting.

Keywords: Procedural city generation, Perlin noise, Performance, Game content

Contents

Chapter 1

Introduction

Exploring a huge open world environment is a desirable feature in games. But creating a big open city such as in the Grand Theft Auto[26] series and Batman: Arkham City[27] involves years of work for a lot of people. Making big open cities in games is simply not feasible for smaller game companies. These games all have great success with their big open worlds making a feasible generated city an attractive technique for smaller companies to be able to use big open worlds of their own.

Content creation requires time and resources; these are both scarce, which is why game companies may want technology to help with this process. To create large amounts of content without a big workforce algorithmically based solutions exist, Procedural Content Generation (PCG). PCG was in the past used to minimize the disk space required for games. .kkrieger[34] is an example of this. It has since evolved into a method to minimize workforce required for content. No Man's Sky[11] is an example of a game using PCG to minimize workforce while maximizing content.

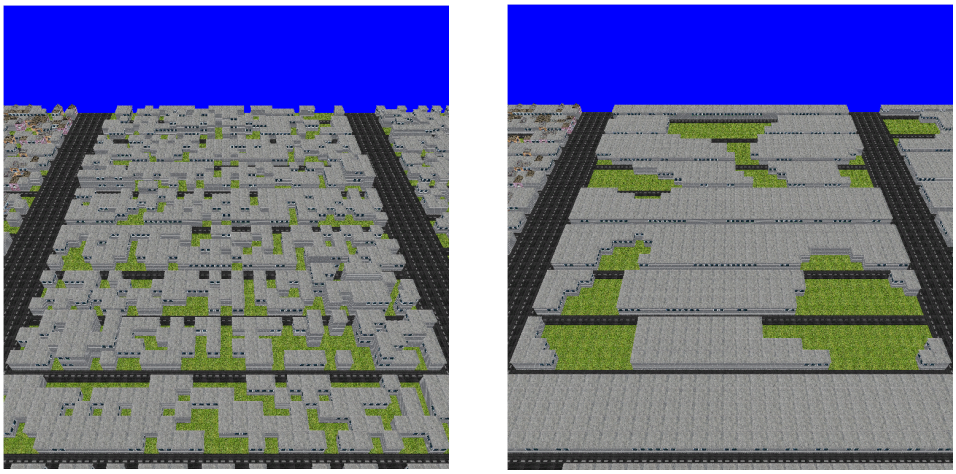
Procedural content generation is a large subject covering many different techniques. Games such as No Man's Sky[11] are using PCG to generate an entire universe of 18 quintillion[19] different planets complete with plants, animals, animations, biomes and sounds. While other games such as Borderlands[10] uses PCG to generate over 17,750,000 different guns[3]. Civilization IV[9], Minecraft[18] and Spelunky[8] are examples of games generating different game environments both in 2D and 3D using PCG. Minecraft's world being eight times bigger than the surface of the earth[17]. With large amounts of content like in these examples it is understandable why handcrafting content would not be feasible and why PCG is an attractive creative method. A lot of work has been done in the field of PCG and it has been used within games for a long time. One of the earliest examples of a game using PCG is Elite[6] from 1984. But no singular PCG solution exists, this may be attributed to the vastly different areas of use.

In this thesis, the possibility of procedurally generating a city with Perlin noise, that would be viable within a game, is investigated. An implementation attempting to create such a city will be made and evaluated through a user study along with data collection from the implementation.

1.1 Background

Perlin Noise is the PCG technique chosen for every stage of the city generation. This technique was mainly chosen because it is feasible to implement within the time frame of this bachelor thesis and implementation time is very important in game development. There are many ways to procedurally generate a city, but many of them are very complicated which often means long implementation time. From a developer's perspective there is not much use for PCG if the implementation time is equal or greater to the time it would take to handcraft every single element.

Perlin noise generates noise with properties that may be interpreted as more natural than purely random. The noise has a natural form, making it a useful technique when generating natural looking content. The difference between random and Perlin noise is illustrated in Figure 1.1.



(a) Buildings placed using *rand*. (b) Buildings placed using Perlin noise.

Figure 1.1: Difference between Perlin noise and rand.

1.2 Procedural content generation

Procedural content generation in games is defined as "[...] *the algorithmic creation of game content with limited or direct user input*"[35]. This is the definition used in this thesis. But PCG has been used for many purposes by many different people. There is no clear definition of exactly what PCG is that is universally agreed upon[35].

PCG is also used in areas outside the gaming industry. SpeedTree[30] is using PCG methods to generate trees that are used in movies such as Avatar[31]. This thesis will focus on the use of PCG in games with the previous mentioned definition.

1.3 City generation

A city has many different definitions in the real world. But a city viable in games is unlike real cities. They are not a large human settlement with governments and sanitary systems. The definition of a city explored in this thesis is "*An area occupied with different kinds of houses connected through a road system.*" Unlike a real city, a city viable in games do not need logic, electricity connections nor gas stations. It simply needs to be an engaging place to experience a game in. With this great simplification, generation of such a city becomes far less complex. To generate a city viable to use in games, three different generation stages have been recognized: Districts, blocks and buildings. These generation stages are used in the implementation.

1.4 Problem statement

How fast the content is generated and the quality of the generated content are the main problems this thesis investigates. To create a city purely using Perlin noise and test if it is viable in games is explored.

1.5 Objectives

Generating a city using the PCG technique Perlin noise and testing the viability in games is the goal. The time to generate a city and the cities viability within games is the corner stones explored in this thesis. The creation of an implementation is necessary as to the best knowledge of the authors there exists no such solution available. There exist city-part creation solutions using Perlin noise such as city boundaries creation, but no cohesive solution creating every part of the city using Perlin noise [33]. The following things need to be completed for the objective to be achieved.

- **Perlin noise:** Everything should be generated with Perlin noise. With only a single PCG technique to implement, the implementation should be fast to create.
- **Implementation:** An implementation that generate viable cities. This implementation will have a user interface that allows the user to rapidly generate many different cities.
- **Data collection:** The implementation must calculate and save correct data that can be analyzed. Loading times and how many times Perlin noise is used for the generation are two types of data that are relevant.

- **User study:** After the implementation is done a user study will be conducted. This user study must have atleast 20 participants answering questions from a questionnaire.
- **Conclusion:** With the data from the user study and the implementation the conclusion that Perlin noise can generate cities viable in games can be done.

1.6 Research question

Can Perlin noise be used to procedurally generate a city viable in games?

Chapter 2

Related work

Procedural city generation has been done many times with different angles. Greuter et al[32] created a method to procedurally generate "*pseudo infinite cities*". This is done by placing the buildings along a grid. The position of each cell in the grid is hashed into a unique number. With this number the building is procedurally generated to be unique from all the other buildings. Because the position of the buildings decides how they look you can walk for a pseudo infinite time through this city, without ever encountering a building that has been seen before. To be able to render such a city, culling of geometry as well as controlling caching (loading and unloading of assets) is important[32].

Müller proposes a method to procedurally generate a city using extended L-systems for the roads, rule based subdivision for the lots and CGA shape grammar for the geometry[21]. This is done in stages; first the roads are generated. The roads naturally generate blocks of land. These blocks are by rule based subdivision divided into lots which fits a house generated by shape grammar as the last stage. This was done in 97.000 lines of code over 6 years of working time[20][21].

Müller et al procedurally generate a 4D city, with the fourth dimension being time. This is also done through L-systems and shape grammars. The growth of the city can be seen through time. Trying to achieve a realistic growth of a city was important here as the results are compared to real life data of city growth[1]. There exists more work where cities are simulated through time[15].

Real time generation and rendering of infinite cities using the GPU has also been done[16]. This city is created with shape grammars. With this technique thousands of building can be rendered at 100 frames per second. Even when the camera is moving fast through the city. This is achieved with advanced culling techniques, frame-to-frame coherence, buffer management and moving important work from the CPU to the GPU[16].

Both L-systems and CGA shape grammars for city generation have updated versions that can be used for future work[23][22].

Ken Perlin's Perlin noise was created to be used as textures for modeling. Since its creation, it has been updated by Perlin himself and has been widely used for other means than textures such as traffic generation[13][12].

Chapter 3

Theoretical framework

3.1 Noise

Lattice noises are simple and efficient ways of generating noise[7]. The idea behind them is to divide a coordinate system into sections spanning between every integer coordinate. These sections are called the integer lattices. By first generating pseudorandom numbers at every lattice and then interpolating between them, noise is generated.

Value noise is a type of lattice noise. It uses a value between minus one and one at every lattice point and interpolates between them. The key difference between value noises is what type of interpolation is used. Methods such as linear and cubic interpolation have been used[7].

Gradient noise differs from value noise by generating gradient vectors at each lattice point, instead of raw values. To calculate the noise value of a point, three steps are taken. First, a vector is calculated from each lattice point to the point. Then, the dot product between this vector and the gradient vector at the corresponding lattice point is calculated. Finally, the noise value is calculated by interpolating between the dot product results.

3.1.1 Perlin noise

Perlin noise is a type of gradient noise. It was developed by Ken Perlin in 1983 and improved in 2002[13]. Perlin noise works in one, two and three dimensions, but for the purpose of this explanation two-dimensional Perlin noise will be used. The algorithm expects two arguments, an `x` and a `y` coordinate. These values are normalized so as to reside between two integer lattices, creating a point. A vector from each lattice point to this point is calculated. One of the gradient vectors is chosen at random based on the normalized coordinates. The dot product between this gradient vector and the vector to the point is calculated at each lattice point. The final value is calculated by interpolating between these dot products. Perlin noise interpolates between values using a proprietary ease curve called fade. It has the form $6t^5 \quad 15t^4 + 10t^3$. Throughout our implementation we used two-dimensional Perlin noise.

3.2 Online vs offline

There are two ways to use PCG algorithms, either offline or online. Online is when the content is generated while the game is being played or in a short loading screen just before the player can start playing. This allows content that is adapted to the individual player and semi-infinite content[4]. In the game Left 4 Dead[36] PCG is used by analyzing the players behavior and altering the experience. This could be considered a mixture of artificial intelligence and online PCG.

Offline generation is when the content is being generated before the player starts playing. This could be done by the developers before releasing the game. Offline generation is useful when generating complicated content that is too slow to generate online. These are the definitions used in this thesis.

3.3 Viability

When developing the implementation, the following requirements were set to ensure that the result is viable in games.

- **Validity:** The city does not have to be realistic but the city must be credible for the player to enjoy the environment. The player should not lose immersion because of broken geometry within the city. The city should also look better than a city generated a random. If a city is prettier and looks more natural than a city generated at random, it is considered valid.
- **Determinism:** The generation should be deterministic. This means that the exact same city can be generated again with the same seed. Nothing should be purely random. This is important so that the user has control over the output of the implementation.
- **Performance:** The city should be generated within a reasonable time frame to be viable in an online setting. A game should never have loading screens for a long period of time. This implementation have a time limit of 15 seconds as Paradox Interactive AB a game company set this as a time limit in Melins and Bengtssons collaboration about procedural generation with the company[2]. The implementation must be viable to use in an online setting as defined in 3.2.
- **Flexibility:** The implementation should be able to generate many different cities with different looks.

PCG can be either *feasible* or *infeasible*. For the generated content to be considered feasible it must fulfill all the constraints[14]. Constraints are game specific but the criteria is that with all constraints fulfilled the game should be playable. An example would be that a player should be able to travel to any location in the city.

Chapter 4

Method

Most related work focus on one of two things: procedurally generating unique building meshes[32] or procedurally generating large cities consisting of similar looking buildings[16]. This work focuses instead on the procedurally generated placement of existing meshes in varied districts. The concept of districts breaks up the repetitive and self-similar nature of the city by introducing distinct changes in appearance between them.

4.0.1 Districts

The generation process starts with the generation of *districts*. A *district* is an abstract representation of an area of the city. It controls what type of buildings it contains, the minimum and maximum height of these buildings and how densely populated the area is with these buildings. The *districts* have a semi-random deterministic spread across the city. There are no constraints on the distribution of the *districts*. This means that one seed may produce a city where 90% of the city is made up of one *district*. There are no constraints to prevent this because this would hinder the variability of the cities generated. The *districts* typically have a lot of blocks where no other districts are competing. But at the borders between districts there is a lot more variation, this is referred to as *border competition*. Here a single block may contain buildings from all three *districts*. This was done to avoid hard borders between the *districts*. The hard borders were considered to look strange and may compromise the cities viability in games. The *districts* were deemed important as they provide variation throughout the city.

4.0.2 Blocks

The next stage in the process is the generation of *blocks*. In this work, a *block* is defined as an area enclosed by four roads. Looking at aerial photographs of large cities, like the one in Figure 4.1, there is a certain pattern in the way roads are laid out. A few major roads stretches through the city and smaller, perpendicular roads connect them together. It was decided to mimic this style. Since a *block* is another abstract representation of an area, it is actually the roads that make them up. The first step of generating *blocks* is to generate the main roads running

through the city. After the main roads have been generated, smaller roads are generated in between, connecting the main roads together. After all roads have been generated, the city has affectively been divided into a non-uniform grid. Each cell in this grid is a *block*. The size of each *block* depends on what *district* it belongs to.

Figure 4.1: Aerial photograph of New York.

4.0.3 Buildings

The final stage is the generation of *buildings*. Each *building* is made up of three meshes: the bottom, middle and top mesh. By dividing a building into sections, it is possible to assemble different pieces and generate new *buildings*. This is a powerful way to use combinatorics to generate more content. For example, creating two whole *buildings* would results in only two unique buildings. Whereas creating two bottom, middle and top sections results in eight possible combinations of *buildings*. Upon generation of a *building*, the attributes of its *district* are taken into consideration. What sections to combine, the minimum and maximum height as well as the density of *buildings*, is all dictated by the *district*.

4.1 Implementation

To explore the procedural generation of a city, a desktop application was created. The application provides the end user with an interface that allows for the configuration of parameters along with the collection of statistics.

4.1.1 Interface

The core functionality of our application is in the form of a homemade static library called Elicras. It was decided early on that there should be a separation between the graphical user interface and the functionality of the application. It is therefore possible to move Elicras into another front facing interface without any significant changes to the underlying code. Elicras is divided into three areas: Rendering, Assets and PCG. Rendering, as the name implies, handles all the interfacing with the graphics card. Assets handles all the loading and using of models and textures. PCG handles all the generation of procedural content. The front facing system interacts with these subsystems through Elicras.

The graphical interface is implemented using Qt[24]. Qt is an abstraction layer between its supported platforms and their user interface. Using Qt it is possible to create an application once and deploy it to all of its supported platforms, including Windows, OSX and Linux. It abstracts away a lot of low level platform code and provides an easy to use editor (see Figure 4.2) where you can create the graphical interface for you application[38][5].

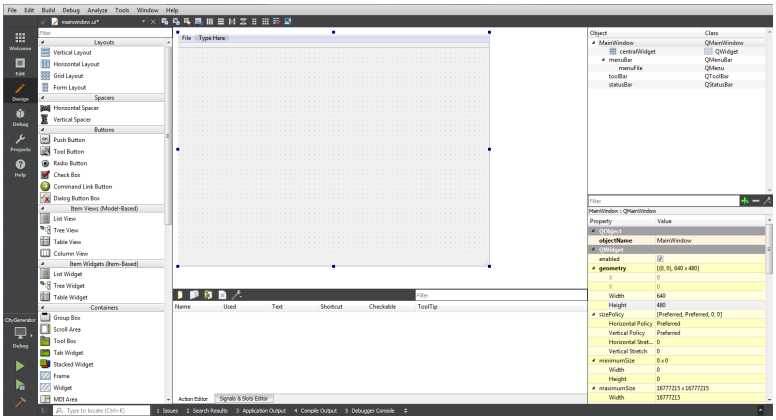


Figure 4.2: Qt Editor.

The interface is divided into three sections that are laid out horizontally (see Figure 4.3). The left section is for statistics such as number of buildings in each of the *districts*, number of roads and the time it took to generate the city. The middle section displays the generated city. The right section of the interface contains controls for altering how the city is generated. There are four parameters per *district*: minimum height, maximum height, density of *buildings* and block size. There is also a global parameter called the seed. The seed can alter the look of the city without altering the per-*districts* parameters. When the end user is satisfied with the parameters, they press the 'Generate' button and a city is generated and displayed on the screen.

Figure 4.3: Elicras graphical interface.

4.1.2 Modified Perlin noise

Depending on the *frequency* and *modifications* of the noise the results can be very different. The steps at which the data is sampled is what the *frequency* is. If the results of the Perlin noise is changed further this is what *modifications* is. The modified Perlin noise is used when generating the *districts*. This noise has much sharper changes in values than regular Perlin noise. Perlin noise without modification returns a value close to 0.5 most of the time. This poses a problem when generating certain things. The *districts* core position for example should have some spread across the city and not be close to each other. The Perlin noise is modified in the following way when generating a *district's* X-axis position; and used similarly for other parts of the generation:

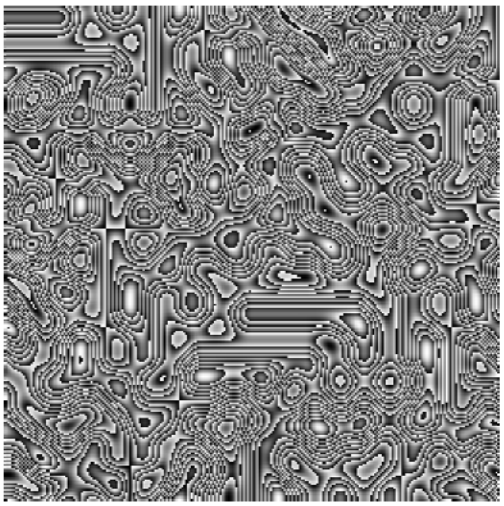
```
   float noise = 20 * this->noise->generate(1.13 * x, 1.22 * y);
noise = (noise - floor(noise));
```

The *frequency* of this modified noise is 1.13 on the X-axis and the *modification* is `noise = (noise - floor(noise))` [29]. The difference between modified and not modified Perlin noise is visualized in Figure 4.4.

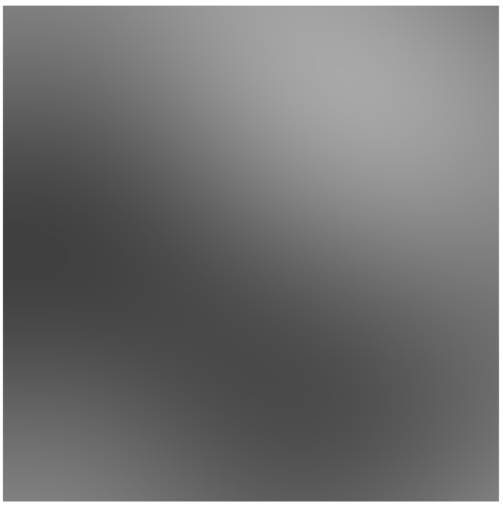
4.1.3 Districts

The *district* PCG generates a 2D grid with cells where each cell contains a *district* number. Each cell is in the later stages filled with grass, roads or buildings. Each cell within the grid represents an area of land in the city. These areas are square shaped with a length of 10 meters which equates to a total area of 100 square meters for each cell.

Each *district* has a starting position; this will be referred to as the *district's* core. The core position of each *district* is semi-randomly distributed throughout



(a) Modified Perlin noise.



(b) Normal Perlin noise.

Figure 4.4: Different uses of Perlin noise.

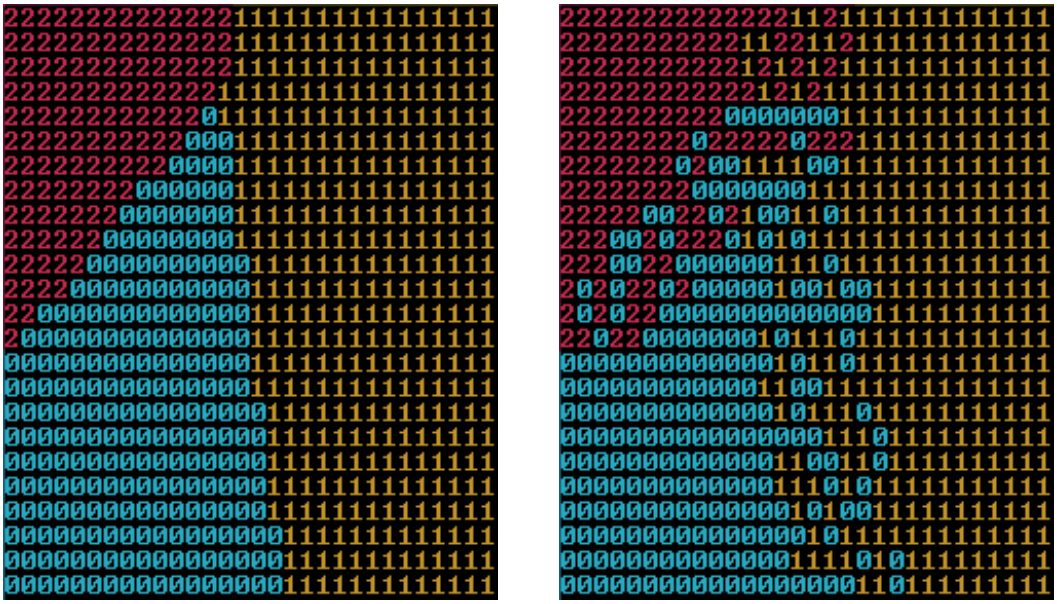
the city with modified Perlin noise. From the core position each *district* takes control over all the cells they are closest to. The distance is calculated with Euclidian distance squared. The squared distance is used because this is faster for the computer to calculate than the real distance. The equation where **A** and **B** are 2D positions:

$$EuclideanSquaredDistance = (B1 \quad A1)^2 + (B2 \quad A2)^2 \tag{4.1}$$

After the *districts* have expanded from their core position, the borders are clear to see. There are often clear lines where on one side there are skyscrapers and on the other there are industry for example. To mitigate this effect *border competition* was implemented. The algorithm finds every cell where two different *districts* meet (i.e. One *district* in the cell and another *district* in the cell to the right). The competition range is 30% of the total grid size. From the border cells we expand half the competition range to the left and the other half to the right; this is done in the X-axis only. For every cell that is in the competition range the two *districts* compete with modified Perlin noise. If the Perlin noise is above the value of 0.5 the *district* to the left wins, otherwise the *district* to the right wins, this can be seen in Figure 4.5. The winning *district* takes ownership of the cell. A visualization of this process can be seen in Figure 4.6.

```
if (noise > 0.5)
    map.at(x, y) = firstDistrict;
else
    map.at(x, y) = secondDistrict;
```

Figure 4.5: Code snippet of border competition.



(a) *District* generation. (b) Border competition.

Figure 4.6: Numeric visualization of the city after *district* generation.

4.1.4 Blocks

After generating *districts*, the roads are generated (effectively creating *blocks*). The first step is to generate the main roads running through the city. Main roads always run vertically through the city, no matter what parameters are used. To figure out where the main roads should be, Perlin noise is used (see Figure 4.7a). By going through the cells of the map horizontally and checking if the Perlin noise for that cell is above a certain threshold, Elicras determines if that cell should be a main road or not. Since the main roads run through the whole city, the algorithm only goes through the first row of cells. If the algorithm determines that a cell should contain a main road, it sets the value of that cell to indicate that it now contains a main road. After generating the main roads, smaller roads are generated. This algorithm works on two levels. The first level calculates the width of the road by starting at its current position in the map and going right until it finds a main road. The next level works much the same way as the algorithm for generating the main roads, but it goes vertically instead of horizontally (see Figure 4.7b). When the second level reaches the bottom of the map, it goes back to the first level. The distance between two smaller roads depends on the *district's* block size. Road generation is complete when the end of the map is reached. The generation of main roads and small roads are shown in Figure 4.8.

```
for (int x = 0; x<WIDTH; x++)
{
    float noiseValue = perlin(x / width, y / height);
    if (noiseValue > MAIN_ROAD_THRESHOLD)
    {
        // cell is a main road
    }
}
```

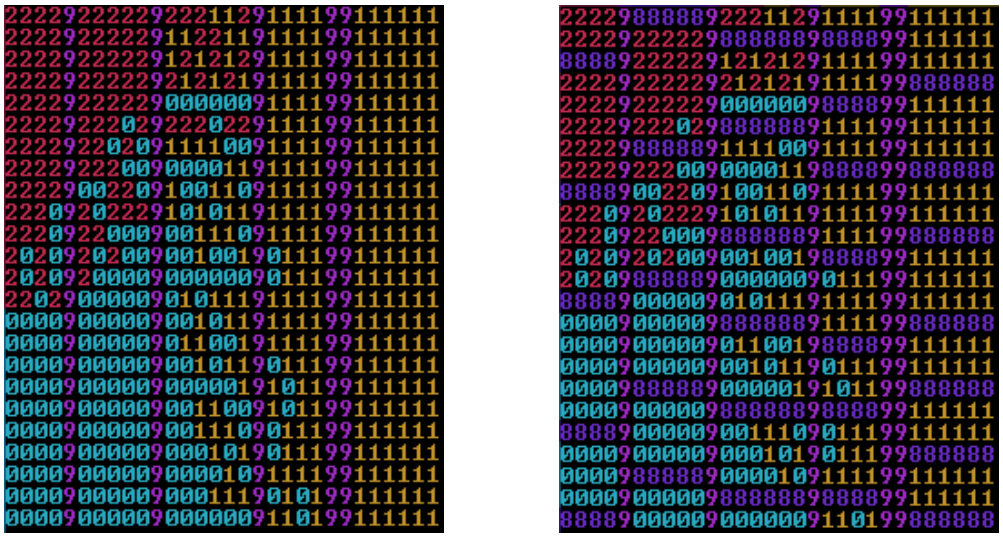
(a) Pseudocode for main road generation.

```
int next = 0;
for (int y = 0; y<HEIGHT; y++)
{
    if (next > 0)
        next--;
    else
    {
        float noiseValue = perlin(x / width, y / height);
        if (noiseValue > SMALL_ROAD_THRESHOLD)
        {
            // cell is a small road

            next = blockSize;
        }
    }
}
```

(b) Pseudocode for small road generation.

Figure 4.7: Psuedocode for road generation.



(a) Main roads.

(b) Small roads.

Figure 4.8: Numeric visualization of a city after block generation.

4.1.5 Buildings

The final stage of the generation process it the generation of *buildings*. At this point the map contains values that indicate whether a cell is a road or a vacant slot belonging to a *district*. This stage goes through the whole map, skipping over any roads. For each cell that the algorithm visits, three of the *districts* parameters are considered: minimum height, maximum height and density. The algorithm retrieves a Perlin noise value for the cell and compares it against the density threshold to determine whether to place a *building* or leave the cell vacant (see Figure 4.9a). If a *building* is placed, the algorithm uses Perlin noise to choose a bottom, middle and top section. It chooses these sections from the list of available sections that belong to the *district*. Perlin noise returns a value between zero and one, this is multiplied by the number of sections (see Figure 4.9b).

```
for (int x = 0; x<WIDTH; x++)
{
    for (int y = 0; y<HEIGHT; y++)
    {
        int district = map.at(x, y);
        const float BUILDING_DENSITY = districtDensities[district];

        float chanceForBuilding = perlin(x / width, y / height);
        if (chanceForBuilding < BUILDING_DENSITY)
            // cell is a building
        else
            // cell is vacant
    }
}
```

(a) Pseudocode for building placement.

```
float noiseValue = perlin(x / width, y / height);

int bottomSection = numberOfBottomSections * noiseValue;
int middleSection = numberOfMiddleSections * noiseValue;
int topSection = numberOfTopSections * noiseValue;
int height = (maxHeight - minHeight) * noiseValue + minHeight;
```

(b) Pseudocode for section selection.

Figure 4.9: Pseudocode for building generation.

The final step of the algorithm determines the height of the *buildings* by multiplying the difference between the maximum height and the minimum height with the result from the Perlin noise and then adding the minimum height. The results of the *buildings* generation is not stored in the map, but in a separate list of building sections that can later be rendered. The placement of vacant tiles is stored in the map. In Figure 4.10 the city can be seen when all generation stages are done. The numbers 0-2 represent what district the buildings there are from, 8-9 are small and big roads and 7 represents the vacant cells.

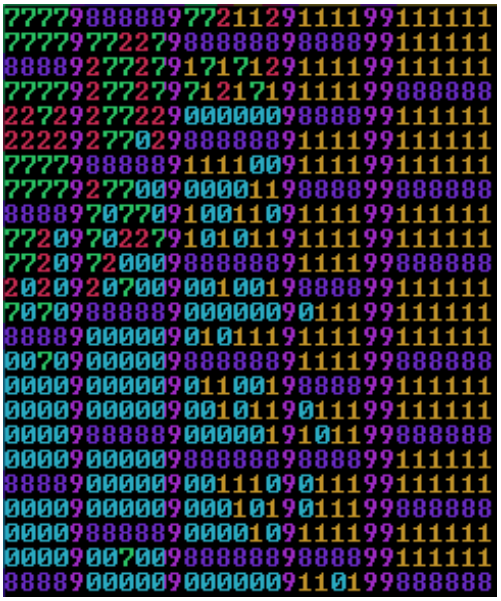


Figure 4.10: Numeric visualization of a city after building generation.

4.2 Data collection

Aside from the core functionality of the application, an effort was made to record any significant data about the generation process. The observer pattern was used at the core of this effort. In Elicras there is a DataManager and many DataHolders. Any class whose data is worth collecting, inherits from the DataHolder base class. This results in a centralized way of collecting data that is easy to use and to setup. When Elicras requests the data from the DataManager, it notifies the registered DataHolders to supply their data. The application gathers the following information about the generation process:

- Number of Perlin noise calls.
- Number of main roads.
- Number of small roads.
- Number of buildings in *districts* 1-3.
- Total number of buildings.
- Number of grass tiles in *districts* 1-3.
- Total number of grass tiles.
- Generation time.

4.3 Experiment Design

The user study conducted to verify the output from the implementation is defined here. The aim is to describe the relevant variables, what was observed, why they were observed and the validity of the experiment.

4.3.1 Introduction

A city generating application has been implemented. This implementation uses the procedural generation technique Perlin noise. Documented city generators use other techniques such as L-systems and often combine several complicated techniques[21]. By using Perlin noise for every part of the city the generation speed is fast and the complexity of the implementation is kept low. However speed is not important if the result is not satisfactory.

4.3.2 Definition

In this user study the viability of the content generated by the implementation was evaluated. The viability in games was decided by two factors. Does the user think the generated cities are viable in games? Do the cities look random or does the participants notice any coherent patterns in the city design?

Object of study: The object of study is the implementation created for this thesis that procedurally generates cities using Perlin noise.

Quality focus: The generated cities viability in games and if the participants find the content to be viable is the quality focus. The perceived structure, the look and randomness was evaluated. A comparison between fully PCG cities, partly PCG cities and fully random cities was made.

Perspective: Five pre-selected images of different generated and random cities was provided to the participant. The images are selected by hand. Description of the five images:

1. Fully PCG city with bad choice of generation parameters.
2. Fully PCG city with good choice of generation parameters.
3. Fully randomly generated city.
4. Fully PCG city with good choice of generation parameters.
5. Districts random but roads and buildings procedurally generated.

4.3.3 Context

The user study was conducted in the facilities of Blekinge Institute of Technology (BTH). The participants were handed a questionnaire explaining what the participant is expected to do. The images of the different cities was also available. All the material for the participants was paper(A4). The same questionnaire and images were available online for participants outside BTH. The objective of this user study is to measure if the participants perceive the implementations generated cities as viable in games. The viability of the procedurally generated cities was compared to randomly and semi-randomly generated cities. This was analyzed to examine if Perlin noise is percieved as more viable in games than random.

4.3.4 Context Selection

Avid gamer vs occasional gamer vs non-gamer: These are the definitions used in this user study, a participant who plays games at least once every week is an *avid gamer*. An *occasional gamer* plays games a minimum of once every month and *non-gamers* play games less than once every month.

Online vs offline: Online data collection is done through the internet. This could be an online questionnaire or a game that automatically collects data while the participants play. Offline data collection is done without the internet. This could be done through interviews or paper questionnaires. This user study collected data Offline through a questionnaire and online through the same questionnaire. The questionnaire can be found at 4.4.

4.3.5 Hypothesis

The hypothesis of the user study is that the participants will notice a difference between a random city and a procedurally generated one. The participants will notice the district structure of the procedurally generated cities and the generated cities will be considered as more viable in games than a random city.

Null hypothesis: Cities generated with the implementation are not viable in games.

Alternative hypothesis: The implementation procedurally generate cities that are viable in games.

4.3.6 Variable selection

The independent variables are the following:

- Building minimum height (1-10).
- Building maximum height (1-10).
- Density (0.0-1.0).
- Block size (1-10).

All the independent variables are connected to different districts and can thereby vary throughout the city. Minimum and maximum height decides what heights all the buildings in that district may be. Density is what decides how many buildings there will be. Block size controls how tightly the roads are placed which in turn decides how big each block in the city is. The dependent variables are if the participant believes if the city is viable in games and if the participants notice any coherent patterns or believes the generated cities are random.

Good vs **bad** parameters: One of the images have intentionally bad parameters. What is meant by this is that the parameters are set to intentionally make a strange looking city. For example, the skyscrapers might have less height than regular houses or the density might be low making the industry district contain mainly grass instead of factories.

4.3.7 Subject selection

The participants were selected by convenient sampling at BTH's facilities. The participants were therefore mainly students. The selected subjects were asked if they want to participate and answer the short questionnaire. The online questionnaire was spread on various social websites such as Facebook.

4.3.8 Design type

The participants sits down and are handed the questionnaire along with the images of the cities and asked to read and answer the questionnaire. There was a

supervisor nearby in case the participants have any questions. The supervisor did not interact with the participants more than necessary. The time of completion is estimated to be five minutes or less.

4.3.9 Instrumentation

The guidelines for the participants is to look at the images of the virtual generated cities and answer the questionnaire honestly. All data collection both online and offline was collected through the same questionnaire, it can be found under 4.4.

4.3.10 Validity discussion

The validity is discussed in three ways: conclusion validity, internal validity, external validity[37].

Conclusion validity: The independent variables that were chosen in the city generation might not be the best variables to generate a city viable in games. The implementation might be the problem and not the technique that is explored: Perlin noise. The sample size of 27 answered questionnaires might not be large enough to draw any conclusion as this might be statistically insignificant.

Internal validity: The images of the different cities were created and picked by the hand. These images might be biased. The perspective these images display the city might not be optimal. For example observed from an angle big buildings blocks the view of the road and this might change the results.

External validity: The subjects in the offline user study were all from BTH's facilities. This means that most of the participants may be students with similar background. Many of the students at BTH also have personal experience with creating games of their own. This may cause the results to be different compared to if the study were conducted in another country at another school. The online study was also aimed mainly at individuals from Sweden. Participants in the online study may be acquainted with the authors of this thesis, as the questionnaire was spread on social websites.

4.3.11 Experiment execution planning

The experiment was executed as follows: In the offline user study, one participant answered the questionnaire at a time. If the participants had any questions a supervisor was close by. The participants were handed the images of the cities along with the questionnaire. The goal was to have at least 20 questionnaires answered fully. A complementary online questionnaire was also available for participants outside the facilities to answer.

4.3.12 Experiment analysis method

The answers from all the answered questionnaires are visualized in several graphs. The images all show a different city generated with a specific set of variables which was mapped to the participants' answers. For example: one city is completely randomly generated. Do the participants notice this or does it look like the generated cities? Is the PCG city with bad parameters perceived as more random than the city with good parameters? The data was analyzed to answer such questions and was then used to draw conclusions about the generated cities' viability in games.

4.4 Questionnaire form

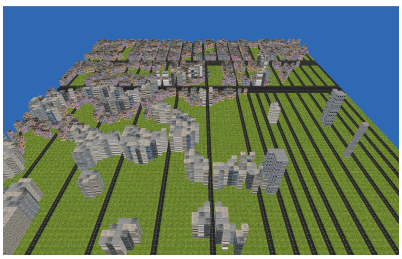
This is a study of how you perceive different generated cities. All answers are completely anonymous. Images of five different generated cities will be shown to you and we ask that you answer a questionnaire about these images. Each image has a number associated with it. Answer the questionnaire by simply drawing a circle around one of the numbers representing the image that you believe is the best answer to the question.

1. How often do you play games? (1 answer)
   At Least once a: Week Month Less than once a month
2. Which city looks the most random? (1 answer)
   1 2 3 4 5
3. Which city looks most natural? (1 answer)
   1 2 3 4 5
4. Which city looks least natural? (1 answer)
   1 2 3 4 5
5. Which city (if any) could be used in a game (0-5 answers)
   1 2 3 4 5
6. Which city looks the most repetitive? (1 answer)
   1 2 3 4 5
7. Which city is the prettiest? (1 answer)
   1 2 3 4 5
8. Which city is the ugliest? (1 answer)
   1 2 3 4 5
9. Which city has the best road network? (1 answer)
   1 2 3 4 5

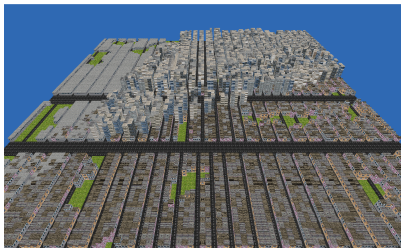
Chapter 5

Results

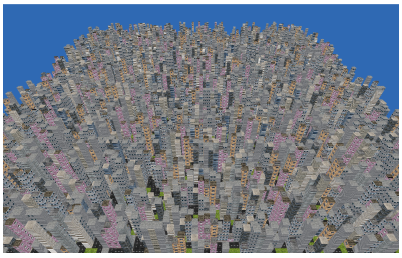
The results from the user study and the implementation are presented here. In order to measure the viability of the cities the following four requirements were set Validity, Determinism, Performance and Flexibility, defined at 3.3. The images used during the user study are shown below. Each city has the size 100x100.



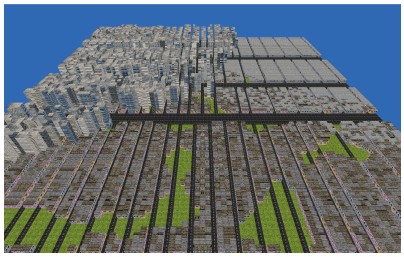
City 1 with bad parameters.



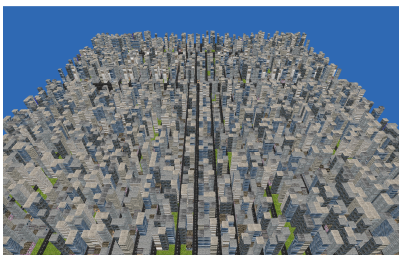
City 2 with good parameters.



City 3 generated at random.



City 4 with good parameters.



City 5 with random districts.

5.1 User study

A user study with 27 participants was conducted. The participants answered questions about images of procedurally generated cities from the implementation created. The user study aimed to measure the viability the cities have in a game.

The participants of the user study are divided into three groups: *avid gamers*, *occasional gamers* and *non-gamers*. By dividing the participants into levels of familiarity with games, a stronger case can be made for the validity of their answers. Figure 5.3 shows that most participants play at least once a week, putting them in the group of *avid gamers*. Not enough participants fell into the group of *occasional* or *non-gamers*. Their results have been discarded, which leaves 25 participants in the *avid gamers* group.

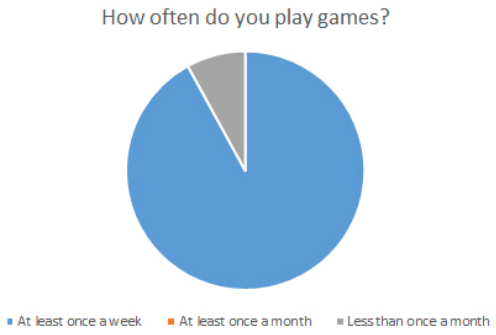


Figure 5.3: Pie chart of how often the participants play games.

The main focus of the user study is to determine if a city generated using our application would be viable in games. The user study asks the participants which (if any) of the cities they think could be used in a game. Figure 5.4 shows the participants answers to this question. This result is represented as a bar chart since this question is a multiple-choice question.

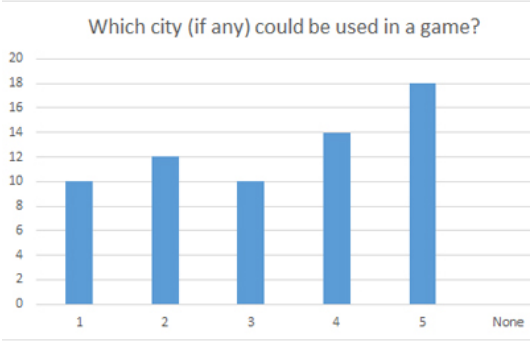
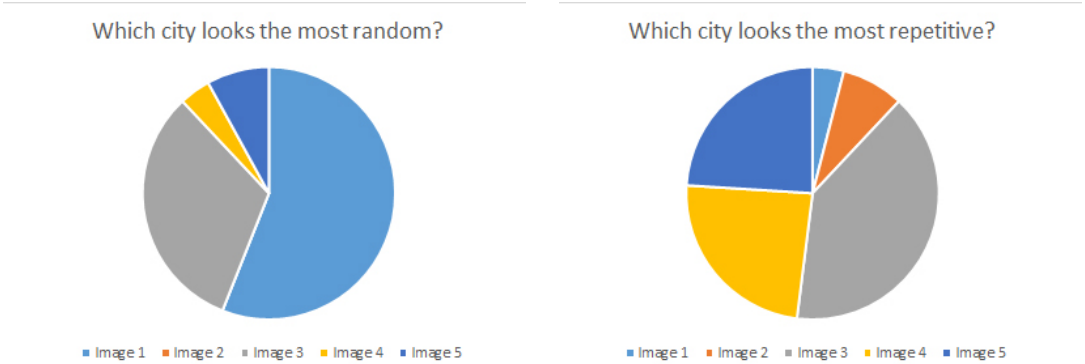


Figure 5.4: Bar chart of which cities the participants think could be used in a game.

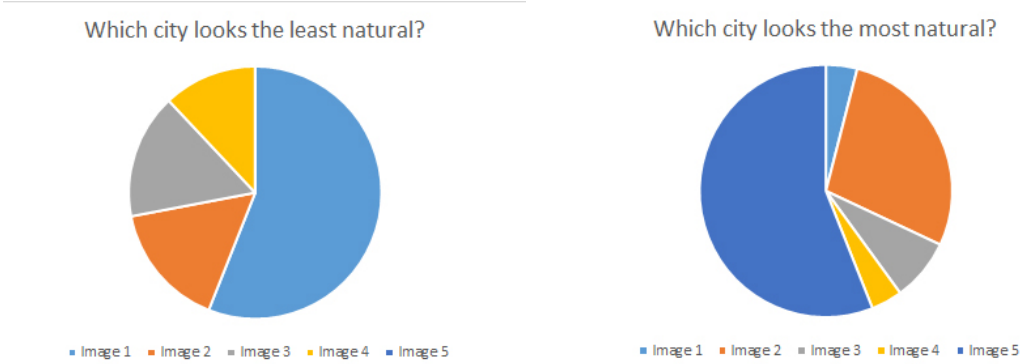
Under Viability (3.3), two requirements are outlined. A generated city must differ significantly from a city generated at random. A generated city must also avoid breaking the users immersion. In an effort to determine if our generation process fulfills these requirements, the participants were asked which city looks the most random and which city looks the most repetitive. The results are shown in Figure 5.5.



(a) Random cities chart. (b) Repetitive cities chart.

Figure 5.5: Pie charts of random and repetitive cities.

Figure 5.6 shows the participants answers to the question about which city looks the least and most natural. This question was intended to clarify the impact of districts on the feel of the city.



(a) Least natural cities chart. (b) Natural cities chart.

Figure 5.6: Pie charts of most and least natural cities.

Another fundamental part of the generation process is the road network. To determine the impact that this stage has on the city, we asked the participants which of the cities had the best road network. Figure 5.7 shows the participants answers to this question.

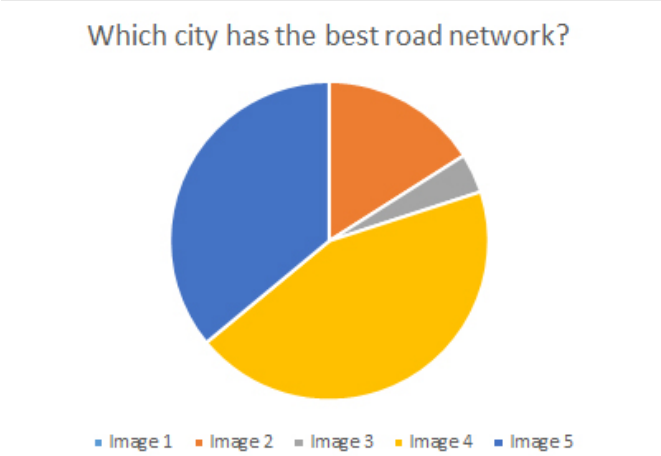
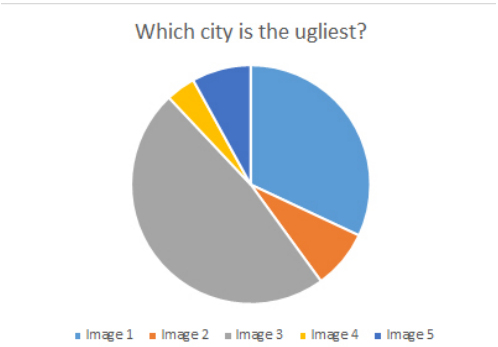
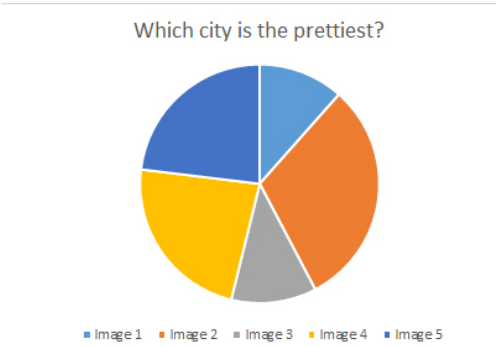


Figure 5.7: Pie chart of which cities has the best road network.

By asking a broad question about the look of the cities and cross referencing the result against their perceived viability in games, a correlation between look and viability can be determined. This is the intention behind asking the participants which of the cities looks the ugliest and which one looks the prettiest. Figure 5.8 shows the result of these questions.



(a) Ugliest cities chart.



(b) Prettiest cities chart.

Figure 5.8: Pie charts of prettiest and ugliest cities.

5.2 Data collection

The following data was collected generating cities using the implementation. The parameters controlling the city were all randomly set during each generation. The cities were generated 100 times for each size respectively. The data shown in the table are all the average numbers of the 100 samples. The generation time is measured in seconds, the size is measured in cells with a width and a height. The *standard deviation* is calculated using the *generation time* data and is also measured in seconds.

Size	Generation time	Standard deviation	Perlin noise calls	Buildings
10x10	0.0017s	0.00009s	155	36
25x25	0.0056s	0.002s	745	213
50x50	0.018s	0.007s	2997	838
100x100	0.065s	0.018s	12083	3534
250x250	0.378s	0.063s	75112	21012
500x500	1.529s	0.209s	302188	87591
1000x1000	6.750s	1.052s	1216841	367258

The data was collected using the following configuration:

Operating system: Windows 7 Ultimate 64-bit SP 1
Processor: Intel Core i7-5820K CPU @ 3.30GHz
RAM: 16,0 GB
Compiler: Visual Studio, 32-bit release mode
Threading model: Single-threaded

Chapter 6

Discussion

6.1 Data and user study analysis

Based on the user study it can be argued that all cities shown are perceived to be viable in games by many, however the study suggest that cities one and three are the least viable as shown in Figure 5.4. City one was created with bad parameters on purpose and city three was completely random. Thus, it is not surprising that these cities seem to be perceived as least viable in games.

6.1.1 City 1

The city generated with bad parameters is the most prevalent in two parts of the user study. As seen in Figure 5.5a the participants in the study found it the most random city. This was unexpected since the most random city is city three, the randomly generated city. It can be seen in Figure 5.6a that this city is seen as the least natural city by a majority of the participants. There is limited evidence, but this may show that procedurally generated cities are highly dependent upon their generation parameters. As the bad parameters generated city are perceived as both the most random and the least natural city. This city is also a contender for the ugliest city, as illustrated in Figure 5.8a.

6.1.2 City 2 and 4

Generated with similar parameters but different seeds these two cities should in theory be very similar. Two cities with similar parameters were included to see if there is any noticeable difference between similarly generated cities. These cities were hypothesized to be the most viable to use in games since they were created using the full procedural generation pipeline with parameters that were thought to be good. Figure 5.4 shows that these two cities are perceived as more viable than cities one and three. This was expected, however, city five seems to be even more viable. Hence, it is possible that the generation pipeline can be improved upon. As illustrated in Figure 5.7 city four seems to have much better road network than city two. Considering that they were generated with

similar parameters but the results diverge to a noticeable degree it may be the case that some aspects of the cities differ more than expected. The difference between these two cities shows that cities generated with similar parameters may be very variable in some aspects even when only small changes are made to the parameters such as changing the seed.

6.1.3 City 3

Randomly generated, this city seems to be the least viable in games along with city one according to the user study, as shown in Figure 5.4. This was expected since procedurally generated content should be more viable than randomly generated content. Where city three is the most prevalent is in the repetitive cities and ugliest cities charts, illustrated in Figure 5.5b and 5.8a. This city is perceived as the second most random city after city one as presented in Figure 5.5a. The results that this city looks the most repetitive to the participants is unexpected. The least coherency and most variance in buildings should be in city three as there is no districts to control the buildings nor road variables. To analyze why city three is perceived as repetitive even though it should have the most variety is outside the scope of this thesis.

6.1.4 City 5

Presented in Figure 5.4, city five is the most viable city for games according to the user study. This is the city with randomly generated districts but otherwise use the same pipeline as cities one, two and four. Randomly generated districts affect where the different buildings may be placed along with the block size. This may result in layouts such as a house surrounded by skyscrapers. This city was seen as the most natural city by most participants, this can be seen in Figure 5.6b. The participants found this city to have the second best road network after city four as shown in Figure 5.7. This may suggest that the participants appreciate some randomness in the city but with the parameters the districts provide the house and road generation. This further supports the idea that the pipeline used in the implementation can be improved upon as suggested in 6.1.2.

6.1.5 Performance

From section 5.2 all the data collected can be seen. The generation time for 100 cells is faster than two milliseconds on average when generating the 10x10 cities. And when generating the 1000x1000 cities the generation of 100 cells is faster than 0.7 milliseconds on average. This may be a viable generation speed in an online setting. As games often aim for 60 frames per second which is 16 milliseconds, a speed of two milliseconds might therefore be viable. This generation speed is without optimizations and on a single computational thread. There are

examples of impressive optimizations for city generation which shows that much optimization is possible [16]. The standard deviation on the biggest city is over one second, the high standard deviation may be caused by the differences between the generated cities. One city may for example have buildings in almost all cells where others might have few buildings. This cause differences in generation time.

6.2 Implementation analysis

The implementation has a number of major limitations. In the interest of overview, a list of limitations that will be discussed is provided below.

- Maximum of three districts.
- All roads are straight.
- One-to-one relationship between cell and building.
- Low quality models are used.
- Flat terrain.
- Arbitrary noise value manipulation.

The implementation was designed from the beginning to only support three districts. This means that the implementation might not work with more districts. Limiting the number of districts to three was an effort to minimize workload while still generating interesting intersections between districts.

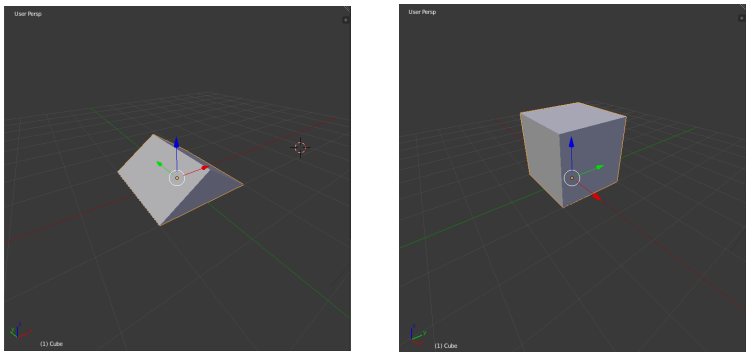
Any city generated using Elicras is based on a two-dimensional grid. A cell in the grid represents a building, road or patch of grass. Depending on the model assets used when generating a city, this can look rigid and unnatural. This is the reason why all the roads are generated in straight lines. In a more believable setting, roads would be generated using curves[28].

Each cell contains at most one building. The implementation has no support for buildings that span across cells. This means that large buildings like warehouses or hangars are difficult to represent. One way to work around this limitation is to consider each cell as a large space of land. By only using small parts of the cell for small buildings and large parts of the cell for large buildings, it will appear as though there is a difference in size between cells.

The models used in Elicras are of low quality. For most building sections simple shapes were used (see Figure 6.1). It is likely that the overall visual fidelity of the city increases with higher quality models.

Any city in our application is generated on a flat surface. Elicras makes no effort to consider the underlying terrain of the city. This renders our implementation ineffective in games with anything but flat terrain.

Perlin noise has been used at every stage of the generation process. There are parts of the process where we have shoehorned the noise function into our



(a) Top section of a building. (b) Mid section of a building.

Figure 6.1: Building sections made up of simple shapes.

implementation rather than using it to its full potential. Examples of this is when Elicras determines where to place roads or what sections to use for a building. In these areas Perlin noise is used like a pseudorandom number generator(PRNG). By using it this way, some of the desirable features of Perlin noise are neglected. In these cases, using an actual PRNG like *rand*[25] might produce similar results, without the implementation overhead. In other cases, the properties of Perlin noise have a visual impact on the result despite being used like a PRNG. An example of this the process of determining whether to place a building or to leave a cell vacant. By checking the noise value against a threshold, buildings are placed in a natural looking way rather than at random (see Figure 1.1).

Chapter 7

Conclusion and future work

7.1 Conclusion

The conclusion of this work is that Perlin noise can be used to generate cities that are viable in games. The believability depends on what parameters are used when generating a city.

7.2 Future work

This section will describe a number of directions for future work.

7.2.1 More generation parameters

In Elicras, only four parameters are available per district. By providing more parameters, the user would have more control over how the city is generated. A few examples are: proximity between districts, size of the city, *frequencies* and *modifications* for Perlin noise (see Page 12).

7.2.2 Mesh generation

This work presents a way of generating a large number of buildings from a small number of parts. This technique is still limited by handmade meshes. By generating meshes from scratch, a larger number of unique buildings can be generated. It would also save time by not requiring handmade meshes.

7.2.3 Curved roads

The generation process outlined in this work requires a two-dimensional grid in which to generate the city. This results in straight roads and rectangular blocks. Generating curved roads would make the city look more interesting and natural. This is another area where procedural mesh generation would be beneficial.

7.2.4 Terrain integration

Any city in our application is generated on flat terrain. A popular application of Perlin noise is terrain generation. By taking the terrain into consideration when generating a city, the two areas of content generation could be combined.

References

[1] Basil Weber and Pascal Müller and Peter Wonka and Markus Gross. Interactive geometric simulation of 4d cities. *Computer Graphics forum*, 2009.

[2] Daniel Bengtsson and Johan Mellin. Constrained procedural floor plan generation for game environments. Master's thesis, Blekinge Institute of Technology, 2016.

[3] Borderlands community. Weapons in borderlands., 2017. Webpage: borderlands.wikia.com/wiki/Weapons.

[4] Chaker, Noor. Procedural content generation in games. *Springer*, 2016.

[5] Dalheimer,Matthias K. *Programming with Qt : Writing Portable GUI applications on Unix and Win32*. O'Reilly Media, Sebastopol, 2;2nd; edition, 2002.

[6] David Braben and Ian Bell. Elite. Acornsoft, Firebird, Imagineer, 1984. Video Game.

[7] David S. Ebert. *Texturing & modeling: a procedural approach*. Morgan Kaufman, 2003.

[8] Derek Yu. Spelunky. Mossmouth, LLC, Microsoft Studios, 2008. Video Game.

[9] Firaxis Games. Civilization IV, 2005. Video Games.

[10] Gearbox Software. Borderlands. 2K Games, Feral Interactive, 2009. Video Game.

[11] Hello Games. No man's sky, 2016. Video Game.

[12] Iria Prieto and Mikel Izal and Daniel Morato. Traffic generator using perlin noise. *IEEE*, 2012.

[13] Ken Perlin. Improving noise. *SIGGRAPH '02*, 2002.

[14] Antonios Liapis, Georgios N. Yannakakis, and Julian Togelius. Constrained novelty search: A study on game content generation. *Evol. Comput.*, 23(1):101–129, Mars 2015.

[15] M. Honda and K. Mizuno and Y. Fukui. Generating autonomous time-varying virtual cities. In international Conference on Cyberworlds, editor, *booktitle*.

[16] M. K. Markus Steinberger. On-the-fly generation and rendering of inifnite cities on the gpu. *Computer Graphics forum*, 2014.

[17] Minecraft Community. Minecraft size of the world, 2017. Webpage: http://minecraft.gamepedia.com/The_Overworld.

[18] Mojang. Minecraft, 2011. Video Game.

[19] No man's sky community. No man's sky planet count, 2017. Webpage: nomanssky.gamepedia.com/Planet.

[20] Pascal Müller. Procedural modeling of cities. *SIGGRAPH '01*, 2001.

[21] Pascal Müller. Procedural modeling of cities part vi. *ACM SIGGRAPH*, 2006.

[22] Pascal Müller and Michael Schwarz. Advanced procedural modeling of architecture.

[23] Peter Wonka and Pascal Muller and Eugene Zhang. Interactive procedural street modeling.

[24] Qt Editor. https://www.qt.io/, 2009. Webpage.

[25] C rand function. Webpage, https://msdn.microsoft.com/en-us/library/398ax69y(v=vs.140).aspx.

[26] Rockstar Games. Grand Theft Auto V. Take-Two Interactive, 2013. Video Game.

[27] Rocksteady Studios. Batman: Arkham City. Warner Bros., 2011. Video Game.

[28] Rahul Sharma. *Procedural City Generator*. IEEE, 2016.

[29] Solarian Programmer. Webpage, Modified Perlin Noise function, https://solarianprogrammer.com/2012/07/18/perlin-noise-cpp-11.

[30] SpeedTree. Procedural tree generation software. Webpage: http://www.speedtree.com.

[31] SpeedTree. Speedtree in movies. Webpage: http://www.speedtree.com/idv-company-profile.php.

[32] Stefan Greuter, Jeremy Parker, Nigel Stewart, Geoff Leach. Real-time procedural generation of 'pseudo infinite' cities. *Graphite*, 2003.

[33] Steven Wijgerse. Generating realistic city boundaries using two-dimensional perlin noise. *University of Twente*, 2007.

[34] .theprodukkt. .kkrieger, 2004. Video Game.

[35] Togelius, J. and Kastbjerg, E. and Schedl, D. and Yannakakis, G.N. What is procedural content generation?: Mario on the borderline. *Proceedings of the 2nd Workshop on Procedural Content Generation in Games*, 2011.

[36] Turtle Rock Studios, Valve Corporation. Left 4 Dead. Valve Corporation, 2008. Video Game.

[37] Wohlin, Claes, 1959- Runeson Per Höst, Martin Ohlsson, Magnus C. Regnell, Björn Wesslén, Anders . *Experimentation in software engineering.* 2012.

[38] Yuan,Feng. *Windows Graphics Programming Win32 GDI and DirectDraw.* Prentice Hall, first edition, 2000.