Surogou

A game world using Perlin Noise

Computer Science, K2 Module, Fall 2014

Anders Bjørn Rørbæk Pedersen (45481, abrp@ruc.dk),
Anders Olsen (45189, andeols@ruc.dk),
Clément Kuta,(55758, ckuta@ruc.dk).

Supervisor: Morten Rhiger (mir@ruc.dk)

December 22, 2014

Abstract

This project is about the design and implementation of the game *"Surogou"* using procedural content generation with the algorithm *perlin noise* to do terrain modelling and distribution of objects. The main objective is how procedural content generation can be utilized in games, while maintaining high performance and to investigate new opportunities that arises, when applying procedural techniques. As procedural content generation is unpredictable by nature, the key challenge in the project is to ensure high consistency and controllability of the algorithm. The project concludes, that utilizing procedural content generation is beneficial in terms of creating a large variety of game spaces procedurally. However, issues needs to be addressed in order to maintain a decent performance. Furthermore, there are several benefits of using these techniques, such as the potential to shorten the development time of a game.

Contents

Chapter 1

Introduction

Early computers had little memory to store data and early video games therefore faced challenges storing large amounts of data, in the form content, such as graphics (textures, sprites), sound (audio) and game spaces (levels and terrain). Some game developers addressed these challenges, not by storing a small and limited amount of content within the game, but by generating content with algorithms as it is needed. In these games the technique and solution to the memory problem was therefore to generate the content in a procedural fashion. In the 1980s, some game developers explored these techniques.

Figure 1.1: *Sentinel* is a videogame, developed by *Geoff Crammond* and published by *Firebird* in 1986, which overcame the limitations of memory by using procedural generated

The videogame *The Sentinel*(1986), had a clever system, to overcome the limitation of the hardware, of using only 48 to 64 bytes to store an impressive 10.000 different game levels [20] while the game *Elite*, from the same year, have a game world consisting of 282 million million million galaxies with 256 solar systems each [19]. Today, videogame developers do not have the same limitations in regards of storage, as hard drives are getting larger in relation to the size of the games. The current size of videogames is described in the French newspaper article *"Le surpoids concerne aussi les jeux vidéo"* [4]. The size of

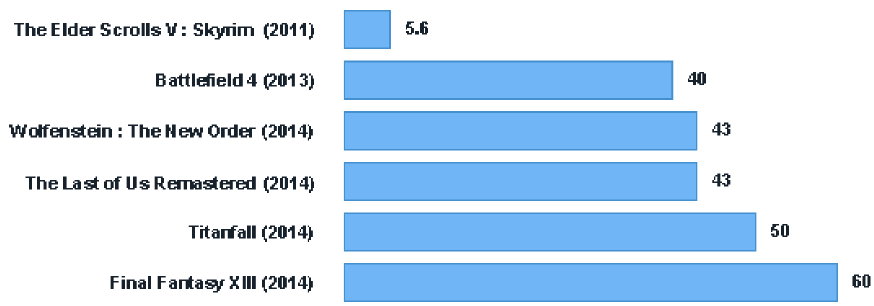


Figure 1.2: Comparison of data storage (GB) between recent video games [4]

the videogames can often be directly related to the amount and size of content. With the new generation of consoles and videosgames, game developers creates large amounts of high definition content manually. This have an direct affect on the required storage space, but more importantly it prolongs the development time. While. The challenges in the early videogames were related to the hardware limitations of storage, whereas videogames of today's challenge is the amount of content, that is being produced for the games. One solution, to the prolonged development time, could be to generate some of the content procedurally. Furthermore, faster processors and larger memory allows new opportunities to generate content procedurally as a way to solve this issue, but also enables the creation of even larger worlds, such as seen in the upcoming videogame *No Man's Sky* (2015).

Figure 1.3: *No Man's Sky* is an upcoming videogame, developed by the British studio *Hello Games*, which features a procedurally generated open universe

Enhanced replayability of a game, can also be done by having adaptive content. To generate content procedurally is also known as Procedural Content Generation (PCG), and can be defined as:

> *"the algorithmical creation of game content with limited or indirect user input"* - Julian Togelius, Noor Shaker, Mark J. Nelson [13, p. 1]

The key term of PCG is *content*, which refers broadly (but not limited to as explained later) to: graphics, audio, game spaces.

1.1 Project Description

The methodological approach to investigate PCG, as a potential means to solve the earlier described issues, will be by creating our own implementation of a videogame named *Sourgou* that utilizes PCG. We will furthermore also look into the possibilities that PCG enables. We will investigate the following four topics, which will also serve as requirements for our implementation:

- Performance
- Infinity
- Controllability
- Consistency

First, it is important that performance is not reduced, when switching from having large amounts of stored content to procedurally generated content. If it is not possible to achieve the same performance, the benefits of PCG would be less attractive. Second, PCG allows for the creation of completely new types of games, where content is generated as it is being consumed (played) [13, p. 3]. Because of this, PCG can be used for generating game worlds that can be characterized as being infinitely large from a players perspective. Therefore, we want to investigate how and which procedural methods can be used to create such an infinite game world. Thirdly, while content is being generated by an algorithm and not directly by a human designer, it is still preferable to have some degree of control over the generated content. If a game have a procedural algorithm for generating trees, it should be capable of generating a wide range of different types of trees. The designer should also be able to define a certain type of tree within the game using a limited set parameters, while not having to design the actual tree. Lastly, in relation to generating an infinite game world, it is often desirable that the game world is consistent in the term of the content that is being generated. When exploring the game world, the player expects that the same content is being generated when he revisits a specific location in the world. It is also important, that changes or modifications in the world are stored.

Project boundaries

Throughout this project, we will focus on the four primary topics, Performance, Infinite, Consistency and Controllability. In order to do this we will investigate various procedural methods that we want to use for our implementation of an infinite game world. In some instances, we use code examples for further explanation. Furthermore, the concrete video game that we present in this project is implemented using *Unity engine*, whereas other engines, such as the newer *Unreal Engine 4*, is not suitable for our purposes because it has constraints within it that prohibits us from using methods of PCG during real-time generation. We will limit ourself to the two algortimes, *perlin noise* and *L-systems* as our primary methods for generating content.

Chapter 2

Procedural Content Generation

This chapter focuses extensively on what PCG is and how the three aspects (Procedural, Content and Generation) can be defined. PCG is well known by experts to be a rather fussy concept and without clear boundaries. This chapter describes our understanding of what PCG is. This is done with the use of a general definition inspired by *pcg.wikidot.com* [9], which is an online community that focuses on collecting and discussing information about Procedural Content Generation. Furthermore, we go into details discussing what specifically constitutes PCG and what does not [16].

As for a general definition of procedural content generation, *pcg.wikidot.com* states the commonly used definition:

> *"Procedural content generation (PCG) is the programmatic generation of game content using a random or pseudo-random process that results in an unpredictable range of possible game play spaces. [...] procedural content generation should ensure that from a few parameters, a large number of possible types of content can be generated."* [9]

The definition explains the importance of randomness and unpredictability and how few parameters should be able to produce a wide range of possibilities.

If content is created intentionally by a user or if the actions that leads to the generation of content can be predicted by the user, it is often not considered as PCG, even if it is assisted by a procedural techniques [16, p. 2]. In the game *Sim City* (2013) the player can make construction plans for roads and types of urban areas, but the actual placement and types of buildings are done using procedural techniques. It is therefore the algorithm, that places the buildings neatly around the networking of roads, that is the procedural process. A game like *Sim City* would therefore not be considered PCG, as the result of placing roads yields a predictable outcome.

Figure 2.1: *Sim City* is a city building simulation, developed by the *Maxis* and published by *Electronic Arts* in 2013.

In the following sections, we will dissect PCG into the three aspects *Procedural*, *Content* and *Generation*.

2.1 Procedural

The procedural aspect involves the methodological approach when generating content. In the following, we will describe these approaches using *Procedural Content Generation in Games: A Textbook and an Overview of Current Research* [13].

- **Online vs. offline** Content that is procedurally generated can either be *offline* or *online*. *Online* content refers to content that is being generated at run-time, while *offline* content have been generated at development time. *Online* content therefore means that it is generated, while it is being consumed.

- **Necessary Vs. optional** PCG can be used for generating game content that is required for completion of a level, or used as auxiliary content, which can be discarded or exchanged for other content. The primary distinction between necessary and optional procedurally generated content is that necessary content cannot be altered in a way that disable the completion of the game while optional is the rest of the content. These two approaches are simply a reminder that when generating content, using PCG, you might have content that is necessary for completion of the game, which creates more restrictions in terms of how PCG must be implemented. Whereas, if the PCG is optional content in the game it can be implemented more loosely.

- **Degree and Dimensions of control** Even though PCG has elements of randomness it is still preferable to have some *degree of control* over the generated content. This can be done by using a set of parameters that limits the possible dimensions or use a seed for a random number generator. This seed would generate a unique world, which means that if one use the same specific seed again the world will be identical every time a game session is started. This approach is interesting because it to some extend goes against the idea of PCG being defined as unpredictable. The reason for this kind of control, or predictability, is still considered PCG because one do not specifically predict or determine what should be in the world but only understands that the particular seed will generate a specific world.

- **Generic Vs. Adaptive** The distinction between *generic* and *adaptive* approach is that a *generic* approach does not take player behaviour into account when content is generated, whereas *adaptive* generation do take players behaviour into account. An example of a game, which uses the *adaptive* approach is *Left4Dead* (Figure 2.2). In *Left4Dead* (2008) the difficulty of the game is adjusted based on the players actions in order to keep the player engaged. An example of *generic* content is in the video game *Terraria* (2011) (Figure 5.15) which generates a random world without the player input being taken into account. When the player acts in the game by building houses or digging for resources, there is no procedurally generated content from the input of the player.

Figure 2.2: *Left4Dead* is a multiplayer First Person Shooter, developed by *Turtle Rock Studios* in 2008, which features adaptive content.

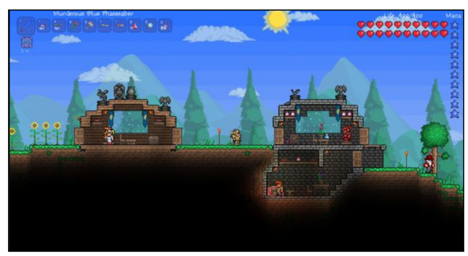


Figure 2.3: *Terraria* is a sandbox games, developed by *Re-Logic* in 2011, which plays in a generic procedurally generated world.

- **Stochastic Vs. Deterministic** These terms define if content is supposed to be recreate-able or random when using PCG. The *deterministic* approach enables the game content to be regenerated given the same starting point and parameters, whereas *stochastic* is the opposite and can be considered completely random. As an example the *deterministic* approach could be the use of a seed for generating the world and the *stochastic* approach could be a consideration of the world being ever changing. In *Diablo III* (2012) (Figure 2.4), the indoor and outdoor maps have rigged starting points and ending points, but the content between them changes regardless of any input from the player. This would suggest a *stochastic* implementation of the content between the starting point and end point of the map.

Figure 2.4: *Diablo 3* is a RPG, developed by the *Blizzard Intertainment* in 2012 were procedural techniques are used for creating the game space.

- **Constructive Vs. Generate-and-test** A *constructive* approach is simply an implementation of PCG which is generated once and is not permitted to be altered. Whereas a *generate-and-test* approach can be generated a number of times until it has reached a satisfactory solution. A *generate-and-test*, could for instance be a recursive backtracking maze algorithm.

- **Automatic generation Vs. Mixed authorship** The approach of *mixed authorship* is an implementation of PCG where a human (player or designer) cooperates with the algorithms in order to generate the desired content. This approach is often used in game development tools such as level design software and 3D modelling software, where unpredictability serves as a source for inspiration.

These approaches are not meant to be determined before making a game, but rather as a tool to understand the different kinds of approaches of PCG. It is also important to understand, that a game is not a procedural game by utilizing PCG, but rather that the content in a game can be procedurally generated. Futhermore, one is not limited to use only one approach, when generating content, but could utilize several.

2.2 Generation

This section describes an overview of the different types of methods for generations of content. This part is concerned with the various types of algorithms that can be used for the generation of content. To describe these methods we use *Procedural Content Generation of Games - A Survey* [7].

- **Pseudo-Random Number Generators (PRNG):** Pseudo-Random techniques are relevant when considering the generation of seemingly random content such as clouds or mountains. *Perlin noise* is a PRNG-based technique because it generates random values for generating noise that can be combined on multiple layers and by scaling.

- **Generative Grammars (GG):** *Generative grammars* are sets of rules operating on individual words or symbols that generate only grammatically-correct sentences. Such algorithms are *Lindermayer-systems* (*L-systems*), which are commonly used for creating fractals and trees, *Wall Grammers* used for building/room generation and *Shape Grammers*. The underlying understanding of GG is that the outcome or end-result will always be a 'correct' result. This means that the generation might use random values but the outcome will always follow a specific pattern and therefore always have in the case of a tree different leafs, roots, body or crown.

- **Image Filtering (IF):** Image Filtering has the main goal to improve an image in regard to a subjective measure or emphasize certain characteristics of an image to display hidden information. The techniques *Binary Morphology* and *Convolution Filters* uses the *IF-based pattern*. These techniques can be used for dilation or erosion of images and remove noise, smooth or sharpen, detect edges of object or even detect the movement direction of objects in an image. These techniques could be used for filtering and manipulating existing textures into new textures, which in turn could save storage space since its generated on the fly and not loaded from storage.

- **Spatial Algorithms (SA):** Is the manipulation of space to generate game content. The output is created by using an input with some sort of structure. This includes techniques for decomposing a map into a grid, integrating grids into maps, which is called layers. It can also be recursive figures that consistently copies themselves like Fractals.

- **Modeling and Simulation of Complex Systems (CS):** Describes natural phenomena with mathematical equations, models and simulations can be used to achieve this. Techniques that focuses on modeling and simulation are *Cellular Automata*, *Tensor Fields* and *Agent-based Simulation*.

2.3 Content

The purpose of this section is to determine what content is in reference to PCG. To fulfil this goal we use the article *"Procedural Content Generation for Games: A Survey"* [7] which classifies the different types of content. The reason we use this particular article's classification of game content is that it was specifically designed for understanding what can be considered to be procedurally generated content in games. The article presents six categories, which are *game bits*, *game space*, *game systems*, *game scenarios*, *game design* and *derived content*. The most fundamental content is called *game bits*.

> *"Game bits are elementary units of game content, which typically do not engage the user when considered independently [...] Textures are images used in games for adding detail to geometry and models, and for giving a visual representation to game elements such as menus. [...] Sound [...] is used to set game atmosphere and pace, and sound effects are used to give feedback to the player on actions and environment change. [...] Vegetation is used in many games for a more realistic and thus immersive look. [...] Buildings are essential to represent urban environments in games. [...] Behavior is the way in which objects interact with each other and the environment, [...] Fire, Water, Stone, and Clouds are often used in games to create a more believable world."* [7, p. 4-5]

The first *game bit*, which is mentioned, is *textures*. There are numerous techniques for procedurally generating *textures*, which includes noise or pattern based algorithms. Noise algorithms are PRNG-based techniques which can be used to create natural looking textures, such as water, clouds and fire, while pattern-based algorithms can be used to create new textures based on existing images or patterns. PRNG and pattern-based algorithms can also be used for creating raw *sound*, while generative grammars can be used to generate a rhythms and music through a rule-based system specified by a composer or sound modular. An example of *vegetation* can be plants and trees that cover ridges of mountains. Vegetation does not only serve as aesthetic features in a game, but can also be a gameplay element, where vegetation then functions as a hiding place for the player. There are various algorithms that enables the generation of vegetation and the algorithm may be specific to a certain type of vegetation. *L-systems* or Lindenmayer systems are popular choices for generation of plants and trees. *Buildings*, which are *game bits* that is often found in games that has some kind of urban environments. Buildings are usually significant to the player, due to the gameplay activities that surrounds buildings, such as collecting resources to a warehouse. The generation of buildings can be either an iterative transformation or a set of rules that generates unique buildings. *Behaviour* is the objects interaction with each other and the environment. Behaviour is determined by the objects characteristics and its surroundings. The explosion of fireworks has a seemingly specific pattern but in reality, it will not always be the same. The explosion of fireworks could also be generated through a type of *L-system*. *Fire, Water, Stone and Clouds* are increasingly relevant in games to create realistic and believable worlds.

These *game bits* are all interesting in their own way and represents the fundamental content of a video game. In the next quote, we look into what the *game space* is.

> *"The game space is the environment in which the game takes place, and is partially filled with game bits among which players navigate.*

> *[...] Indoor Maps are depictions of the structure and relative positioning of indoor space partitioned into rooms. Rooms may be interconnected by corridors, overlapped in layers interconnected by stairs, and grouped altogether in dungeons. [...] Outdoor Maps are depictions of the elevation and structure of an outdoor terrain. [...] Bodies of Water such as rivers, lakes, and seas are often used as map obstacles or even as interactive game space. Other map features, such as teleportation areas, etc. and mountains, ridges, ravines, grottoes, etc. may also be part of game space."* [7, p. 5]

Indoor Maps typically consist of rooms that are interconnected by corridors. A simple implementation a maze could be done using PRNG-based or search-based algorithms. It is common for games to have both *Outdoor Maps* and *Indoor Maps.* Outdoor maps can be represented by elevation maps, known as height-maps (which are greyscaled textures) which can be generated with noise algoritmes, such as *value noise*, *perlin noise* (PRNG-based) or *worley noise* (CS-based). *Game space* also includes, the manipulation of height-maps, using IF-based algorithms for simulating erosion, which can create be utilized to generate *bodies of water*, such as rivers, lakes and seas. Another approach to generate outdoor maps is by using agent-based algorithms, where a landscape is carved out by multiple agents, which each has a specific role.

> *"The use of game systems can make games more believable and thus appealing. [...] Ecosystems govern the placement, evolution, and interaction of flora and fauna through algorithms and rules. [...] Road Networks form the basic structure of an outdoors map, serving different purposes such as transportation between points of interest, and structuring of and transportation within cities. [...] Urban Environments are large clusters of buildings where many people live together and interact with their surroundings. [...] Entity Behavior Many types of complex player-environment interaction need to be possible to make the player experience that a virtual world is life-like." [7, p. 5-6]*

Ecosystems, refers to the distribution, evolution and interaction with of vegetation in the *game space*. One way to create an *ecosystem* is to combine several procedural algorithms. *L-systems*, could be responsible for generating the trees, while also be in control of how the tree should grow over time, where parameters, such as the elevation (height-map) of the terrain or the level of moisture in the ground could have an impact on the generated tree. The player could also alter the ecosystem by interacting with it. If the player, were to chop down trees, the moisture levels would change, which would result in muddy ground. *Road Networks* creates structure between points of interest. The main issue when focusing on road networks is getting a balance between randomness and structure. A common use for making road networks is by using *L-systems*, which are used to control parameters, such as population density, road patterns and local constraints (ie. what to do near water, at a certain amount of population and should road patterns be rectangular or round shaped as well). Buildings (which are *game bits*) are usually generated through some pre-defined rules, but these rules can be altered based on the population density, which basically works like an evolutionary system that affect the growth of buildings. A good example of *Entity Behaviour* is procedural algorithms that achieves group movement of entities (ie. humans) in order to create a realistic illusion of life-like content. This could be some sort of Artificial Intelligence technique that makes the entities react and creates a variety of actions, which makes the entities seem intelligent.

> *"Game scenarios describe, often transparently to the user, the way and order in which game events unfold. [...] Puzzles are problems to which the player can find a solution based on previous knowledge or by systematically exploring the space of possible solutions embedded in the problem [...] Storyboards are design aids for the game developer or player. [...] The Story of a game is often key in creating a good gaming experience. [...] The concept of Levels is used in nearly every game as a separator between gameplay sequences."* [7, p. 6-7]

Puzzles in a game are often used in game as a gameplay element, where the player needs to solve a certain problem in order to progress in the game. Examples of puzzles are crosswords, riddles or chess(-like) gameplay. The common understanding of *Storyboards* are represented as comics, which has a sequential scene description of events. However, they can also guide the players the way that *Story* is another example of game scenarios which creates a logical process for game events to unfold. It also provides the player with the motivation to accomplish goals. *Levels* in game are used to get the player from a start position to an end position. When the player arrives at the exit of a level, it will be determined by the game conditions whether the player will progress to the next level, while this also serves as dividing the game into stages gradually increases the difficulty of the game.

> *"The System Design of a game entails "the creation of mathematical patterns underlying the game and game rules" [...] The World Design of a game is "the design of a setting, story, and theme""*

Since the *System Design* is based on underlying mathematical patterns and game rules the main challenge is to provide a balanced game experience for all players involved in the game. This becomes increasingly more important when considering a competitive game such as *Dota 2* (2013). *Dota 2* has a pool of heroes with individual abilities, which the player picks from. All these heroes needs some sort of mathematical balance in order to make a fair game for every player. In terms of a procedurally generated game with a mathematical pattern underlying the game we could consider a generator that takes game-board rules as input and creates a new game by changing or transforming those rules. The *World Design* is very similar to the *System Design* but instead it focuses on the setting, story and theme of the world. The procedural generation would therefore result in a new model of environments or maps.

Figure 2.5: *DOTA 2* is a multiplayer online battle arena, developed by *Valve* in 2013, where teams of online players are competing against each other

> *"News and Broadcasts A game may show its players news items based on their actions and other changes in the game's universe; [...] Leaderboards—player ranking tables—are popular for a variety of game genres and are used by fan-sites to serve millions of players"*

News and Broadcast & *Leaderboards* are considered derived content because it is derived from the players play trough of the game. This content could be used for a procedural generation. A case study [2] on the videogame *World of Warcraft* (2004) resulted in the creation of comics based on key moments in a player game session. They were able to create a system that could collect a set of significant screenshots accompanied by matching comic layout with speech bubbles that mimics sound effects and dialogues. Another example is online leaderboards, which is used to show ranks of a players in competitive games.

Figure 2.6: An automatic comic generation system [2] for the videogame *World of Warcraft* (2004)

2.4 Summary

In summary, not all systems in videogame has to use procedurally methods. An example would be game that has a soundsystem, which does not utilize procedural generation, whereas the textures in the game could be done procedurally. It also is important to highlight that Procedural does not mean random **??**. It is not the content that is random, but the stochastic elements (the random number generator) in the algorithm, that makes the content unpredictable.

Chapter 3

Design of Surogou

In this chapter, we will describe the design of our implementation of a procedurally generated game world. Our focus is not by any means to make a game that is entertaining, but rather focus on how to create the game world, which means that we will not have a typical game design document. That said, we will still have some simple game mechanics, which is exploration of the game world and collection of coins. Our primary focus is on *Game Spaces* and *Game Bits* and the creation of them using PCG methods, as frequently as possible. Therefore, we will go into detail about the design choices in order to explain the chosen methods and algorithms. This chapter will continue in a requirement specification to clarify, what needs to be done in the implementation of our game. Furthermore, we will look into the procedural methods that we have been using to create our game *Surogou*. We will limit ourselves to only explain the applied methods in this project, even though there are several other approaches to create this content. To fulfil that goal, we will first explain how we can model terrain by creating two-dimensional height maps that produces natural looking landscapes. We will also look into the basic concepts of how to create a consistent and infinite game world by using the players position as input.

Design Choices

In reference to *game bits* and *game spaces*, we want to procedurally generate *game bits*, such as textures and vegetations, while our *game space* will be an outdoor environment landscape. We therefore need to identify methods that can produce content such as mountains, ridges, trees, rocks, water, earth etc. We will also focus on the topic of infinity. To define it, one could say that the player should theoretically be able to go infinitely in one direction, where new content will keep being generated. This also means that we need to use procedural methods that allows us to create content in real-time (the online approach) in order to maintain the illusion of an infinite game environment. There are many methods to procedurally model terrain, as described in the *Procedural Content Generation* chapter. We want to create a realistic landscape and the most adequate method for that is to use fractal noise generators and especially *perlin noise*, which can be used to generate a natural environment [12]. Additionally, we can also use *perlin noise* for the distribution of objects such as trees, rocks and coins. As the world should also be populated with vegetation, we will look into *L-systems*, which seems to be the obvious choice for creating organic looking *game bits*, such as plants and trees.

3.1 Requirements

In our project description, we explained the four main topics, which are Performance, Infinity, Controllability and Consistency. We will now translate these topics into requirements for our implementation of a game world, which is named *Surogou*.

- **Performance** is important because the content is generated in real-time which produces heavy and continuous computation of content. The main goal is to have a stable and high number of frames per second to produce a good game experience for the player. It is common for PC and consoles to have FPS in the range of 30 to 60 FPS [15].

- **Infinity** is the concept that we want to implement without impacting the size of the game. This means that content should be produced infinitely as the player progresses trough the world.

- **Controllability** is a requirement, as *"procedural content generation should ensure that from a few parameters, a large number of possible types of content can be generated"*[9], and relates to the degree and dimensions of control of PCG. These parameters therefore serve as a sort of controllability that can generate different types of content.

- **Consistency** is a requirement for our game to produce a logical and coherent game world. That means that whenever the player revisiting a previously seen area in the game world, objects in that world should be consistent, in regards of generating the same content. If the player meets an animal, he would also expect it to be there the next time he revisits that place, while interactions that changes the world, should also remain changed. An interaction could be that the player can slay the animal, which means that he would expect that the animal keeps being dead the next time he revisits that location.

3.2 Terrain Modelling

> *Height-map generation is nowadays often based on fractal noise generators, such as perlin noise, which generates noise by sampling and interpolating points in a grid of random vectors. Rescaling and*

> *adding several levels of noise to each point in the height-map results in natural, mountainous-like structures.* [12, p. 2]

When modelling terrain, certain important properties [13, p. 57] must be presented:

- The realism of the output
- The performance of the algorithm
- The control of the generation process.
- Consistency (should produce the same result each time)

One of the techniques, when modelling terrain is to use *height-maps*, which consists of two-dimensional grids of elevation. There are several procedural algorithms for generating these *height-maps*, but most of them are *coherent noise* algorithms which produces different kinds of noise, such as *value noise* (A), *perlin noise* (B), *simplex noise* (C) or *worley noise* (D).

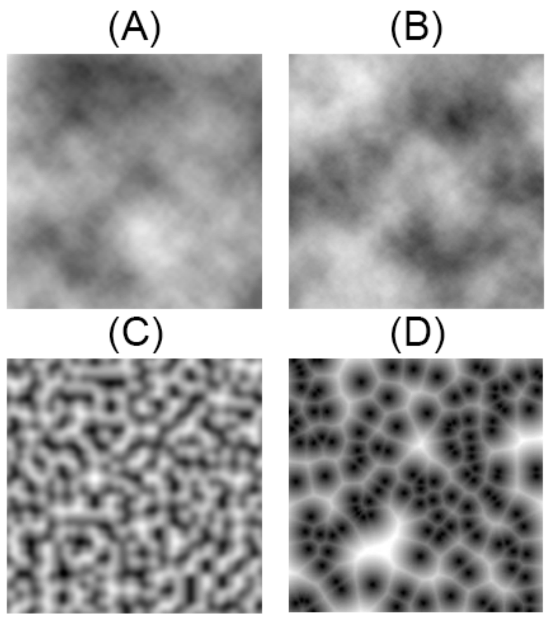


Figure 3.1

Besides the mentioned noise(s), other techniques have also been known to be used. An example is agent-based search algorithms [13, p. 67], where a number of agents are used to carve out the terrain and cellular automata that imitate natural phenomenons, such as water erosion.

Generating a random terrain could simply be done by using a random number generator to determine the heights on each point, and while this technique works, the results are not useful [13, p. 59]. Even though the performance is rather good, the produced output does not look natural, as every value is generated independently. Neither do we have any control of the output. However, the coherent noise algorithms addresses theses issues of control of the generation process and the realism of the output. A *Coherent noise* algorithm can be defined by the following properties [1]:

- Passing in the same input value will always return the same output value
- A small change in the input value will produce a small change in the output value
- A large change in the input value will produce a random change in the output value

All the *coherent noise* algorithms uses *random noise* as a starting point and applies some kind of interpolation to smooth out the values.

Perlin noise is one of the preferred algorithms for creating *gradient noise* for *height-maps* [13, p. 67] and textures such as marble, wood, clouds, fire. *Perlin noise* was first done by Ken Perlin, while working on the film Tron (1982) and can be implemented with an arbitrary number of dimensions, where the two-dimensional version is the one that is commonly used for creating *height-maps*. Three-dimensional versions of *perlin noise* can be seen in the video game *Minecraft* (2009), that creates a voxel-based game world, where three dimensions are required to carve out terrain features, such as caves and ravines, which cannot be represented in a two-dimensional grid.

Figure 3.2: *Minecraft* is a videogame, developed by *Mojang* in 2009 and use *perlin noise* for generating the terrain

In this project, we will be focusing on using *perlin noise* as our preferred method in order to create two-dimensional *height-maps*. Due to the relative simple implementation and the amount of resources and documentation explaining the algorithms in code snippets such as catlikecoding.com [6], which our implementation will be based on (see more on page 58)

The simplest implementation of the *coherent noise* algorithm is a *value noise*. Figure 3.1 (A) gives a good basis on how these algorithm works.

Value Noise

> *The elevation at a specific point on the earth's surface is statistically related to the elevation at nearby points. If you pick a random point within 100 km of Mount Everest, it will almost certainly have a high elevation.* [13, p. 59]

The idea behind creating a *coherent noise* algorithm, is to interpolate between the adjacent neighbour values in a lattice grid to avoid sharp transition. The algorithm can be divided into two parts: creating an array of pseudo random numbers that makes basis for the grid and the interpolation between points. The array usually consists of 511 values ranging from 0 - 255 and can either be done by using a predefined permutation table or created by a random number generator, if seeds are required. Important parameters for the algorithm are the `frequency` and `amplitude`, which determines the space between the samples and the upper and lower bound of the values. When implemented in two dimensions, a low `frequency` with high `amplitude` can produce mountain like shapes and a high `frequency` and low `amplitude` will produce a hill like landscape.

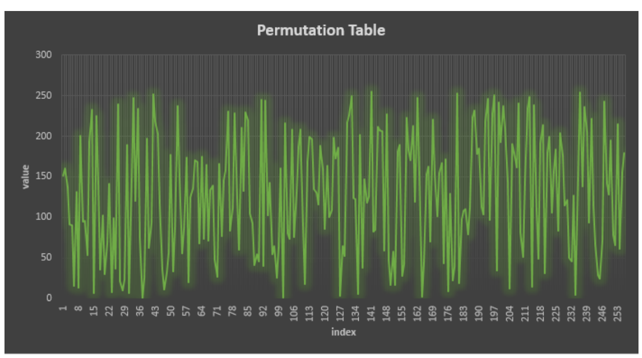


Figure 3.3

The above permutation table (figure 3.3) is the one originally used by Ken Perlin [10] in 1983. This sequence of number will always produce a tilling pattern, but will only be noticeable, if we see a large portion of the array. Whenever interpolating between the values in two-dimensional *value noise* we need to determine the four corners of the sample point.

Listing 3.1: Value Noise Algortime

```
private const SIZE = 511:
private static int[] perm = { 511 values }
public static float Value2D (Vector3 point, float frequency) {
    point *= frequency;
    int ix0 = Mathf.FloorToInt(point.x);
    int iy0 = Mathf.FloorToInt(point.y);
    float tx = point.x - ix0;
    float ty = point.y - iy0;
    ix0 &= SIZE;
    iy0 &= SIZE;
    int ix1 = (ix0 + 1)& SIZE;
    int iy1 = (iy0 + 1)& SIZE;

    int h00 = perm[perm[ix0] + iy0];
    int h10 = perm[perm[ix1] + iy0];
    int h01 = perm[perm[ix0] + iy1];
    int h11 = perm[perm[ix1] + iy1];

    tx = Smooth(tx);
    ty = Smooth(ty);
    return Mathf.Lerp(
      Mathf.Lerp(h00, h10, tx),
      Mathf.Lerp(h01, h11, tx),
      ty) * (1f / SIZE);
    }
```

Listing 3.2: Smooth(float t)

```
private static float Smooth (float t) {
  return t * t * t * (t * (t * 6f - 15f) + 10f);
}
```

We first need to store the lattice coordinates to the sample point in the variables `ix0`, `ix1`, `iy0` and `iy1` and the remaining fractional part in the `tx` and `ty` (Listing 3.1). A graphical representation of this can be seen in figure 3.6 (A). To avoid overflow in the permutation table, we make sure that the values are in range by using the remainder operator. In order to get the values from each corner, we look up the permutation table and store the result in `h00`, `h10`, `h01` and `h11`. The next step is to interpolate between the values to get a single value. The interpolation between the points is done in two steps. First, the fractional part is smoothed in the horizontal and vertical direction to get a weighted average between the points. This interpolation is known as bilinear interpolation, as it is done in two dimensions. Bilinear interpolation though have its drawbacks (see figure 3.4) as *"mountain slopes become perfectly straight lines, and peaks and valleys are all perfectly sharp points"* [13, 60].

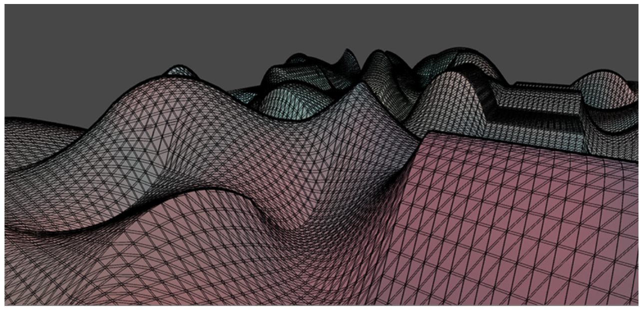


Figure 3.4: Bilinear Interpolation, $f(x) = x$

To avoid this, one can use a different kind of function to smooth out the slope. The function for linear interpolation can be represented as $f(x) = x$, and will produce a straight line between 0 and 1, while $f(x) = 2x^3 + 3x^2$, will smooth out the values in a s-shaped like curve. This is seen on figure 3.5.

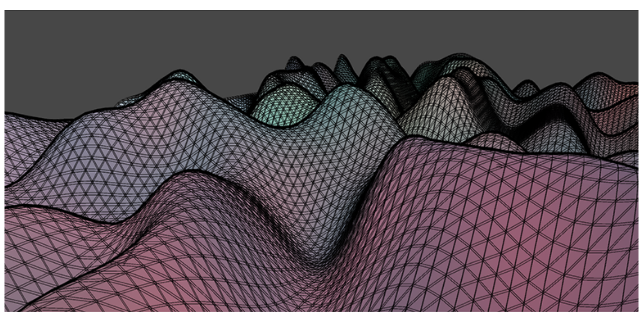


Figure 3.5: S-shaped Polynomial, $f(x) = 6x^5 - 15x^4 + 10x^3$

When smoothing the fractional part, one can use different polynomials. The most commonly used is s-shaped polynomial, $f(x) = 6x^5 - 15x^4 + 10x^3$, and can be seen on figure 3.6 (B). The smoothing part can be seen on line 19 and 20 in listing 3.1 and listing 3.2. The second part is to interpolate between the four corners using the smoothed fractional part as seen on line 21-25 in listing 3.1.

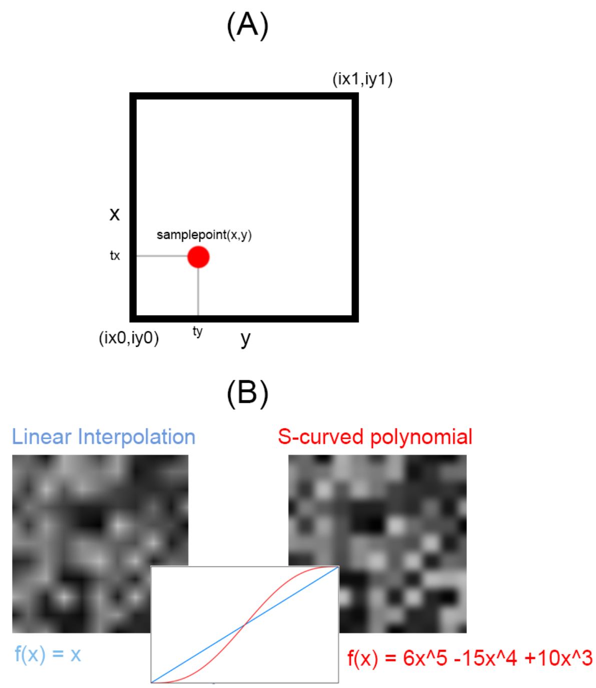


Figure 3.6

Perlin Noise

Perlin noise looks similar to *value noise* but instead of using points, *perlin noise* uses gradients represented as vectors. *Perlin noise* has advantages over *value noise*. One of them is that *perlin noise* creates a more smooth transition of the values, by interpolating between slopes of different steepness and direction [13, p. 61]. This means that our lattice grid is populated with vectors instead of values. As we are using gradients, it is possible to have an extra layer of smoothness. Rather than smoothing the change of values, we interpolate between rates of change of values. Our implementation of *perlin noise* is based on *Jasper Flicks* tutorial [6] on noise algorithms and can be found in the appendix on page 115.

Listing 3.3: Perlin Noise 2D

```
1 const int SIZE = 255;
2 private int[] perm = new int[SIZE + SIZE];
3 private static Vector2[] gradients2D = {
4   new Vector2 (1f, 0f),
5   new Vector2 (-1f, 0f),
6   new Vector2 (0f, 1f),
7   new Vector2 (0f, -1f),
8   new Vector2 (1f, 1f).normalized,
9   new Vector2 (-1f, 1f).normalized,
10   new Vector2 (1f, -1f).normalized,
11   new Vector2 (-1f, -1f).normalized
12 };
13 private const int gradientsMask2D = 7;
14 private static float sqr2 = Mathf.Sqrt (2f);
15
16 public float Perlin2D (Vector3 point, float frequency)
17 {
18   point *= frequency;
19   int ix0 = Mathf.FloorToInt (point.x);
20   int iy0 = Mathf.FloorToInt (point.y);
21   float tx0 = point.x - ix0;
22   float ty0 = point.y - iy0;
23   float tx1 = tx0 - 1f;
24   float ty1 = ty0 - 1f;
25   ix0 &= SIZE;
26   iy0 &= SIZE;
27   int ix1 = (ix0 + 1) & SIZE;
28   int iy1 = (iy0 + 1) & SIZE;
29
30   Vector2 g00 = gradients2D [perm [perm [ix0] + iy0] &
        gradientsMask2D];
31   Vector2 g10 = gradients2D [perm [perm [ix1] + iy0] &
        gradientsMask2D];
32   Vector2 g01 = gradients2D [perm [perm [ix0] + iy1] &
        gradientsMask2D];
33   Vector2 g11 = gradients2D [perm [perm [ix1] + iy1] &
        gradientsMask2D];
34
35   float v00 = Dot (g00, tx0, ty0);
36   float v10 = Dot (g10, tx1, ty0);
37   float v01 = Dot (g01, tx0, ty1);
38   float v11 = Dot (g11, tx1, ty1);
39
40   float tx = Smooth (tx0);
41   float ty = Smooth (ty0);
42   return Mathf.Lerp (
43   Mathf.Lerp (v00, v10, tx),
44   Mathf.Lerp (v01, v11, tx),
45   ty) * sqr2;
46 }
```

The main difference, between *value noise* and *perlin noise* can be seen on line 30-39 (Listing 3.3), where vectors are used instead of points. We still use the same permutation table, but we define the vectors as seen on line 30-34 (Listing

3.3). The lattice grid can thereby be represented as a grid of random vectors as seen on figure 3.7 (B). These vectors are stored in `g00`, `g10`, `g01`, `g11`, and returns one of the eight vectors stored in the `gradients2D` table, which holds a vector with a given direction in space as seen on figure 3.7 (A). To find a value at a non-lattice point, we need the four adjacent neighbours. If one consider only the top left corner, we can calculate a value on that slope, simply by multiplying the distance we have travelled along that gradient, also known as the dot product of two vectors [13, p. 62]. The distance can also be referred to as the fractional part stored in `tx0` and `ty0`. The dot product of each corner gradient is stored in `v00`, `v01`, `v10`, `v11`. Then we repeat the same interpolation between the points, as with *value noise* to get a smooth value.

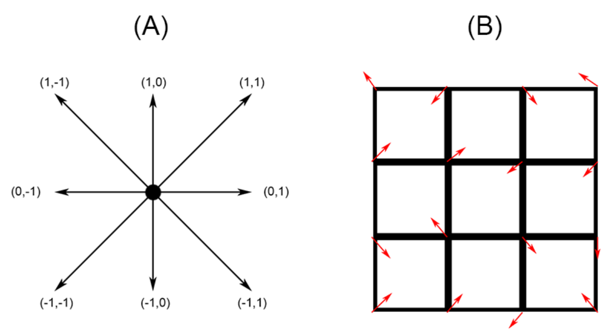


Figure 3.7

Fractal noise

Fractal noise can be produced by combining several layers of noise with different `frequency` and `amplitude` to create *fractal noise*. The first layer is a terrain with large features, we then add smaller features trough each iteration, which are finally added together [13, p. 62]. The number of iterations is controlled with a parameter we call `octaves`. Our implementation uses `lacunarity` and `persistence` [6] to control and change the output of the noise, where each has a different effect on the end result.

Listing 3.4: Fractal noise

```
public float FractalNoise2D (Vector3 point, int octaves, float
    frequency, float lacunarity, float persistence, float gain)
{
  float sum = Perlin2D (point, frequency);
  float amplitude = 1f;
  float range = 1f;
  for (int o = 1; o < octaves; o++) {
      frequency *= lacunarity;
      amplitude *= persistence;
```

```
        range += amplitude;
        sum += Perlin2D (point, frequency) * amplitude;
    }
    return (sum / range) * gain;
}
```

An example code for *fractal noise* can be seen on Listing 3.4, and takes six parameters, which can be used to control the output of the noise. These both apply to *value noise* and *perlin noise*. The first parameter (`Vector3 point`) is simply a vector which, represents a position in the lattice. The number of iterations is controlled by the `octaves` parameter, while `frequency` determines the spacing between the points. `Lacunarity` determines how quickly the `frequency` increases in each iteration, while `persistence` determines how quickly the `amplitude` is increased. As we might want values that goes beyond -1 to 1, we can control this by a `gain` parameter.

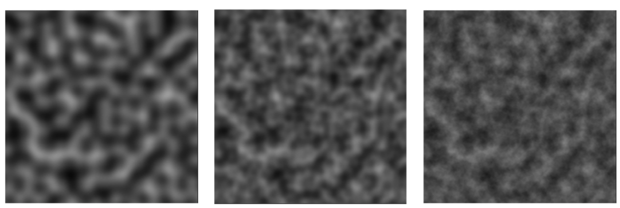


Figure 3.8: Fractal noise with 1, 2 and 8 octaves

Other uses for noise

Besides using noise algorithms for textures and height-maps, one can also use these to distribute vegetation and/or objects around the world, or create different landscape types or biomes by combining several of these noise algorithms with a threshold value. One could have a noise algorithm for the moisture in the ground, where a high value of moisture would create a certain type of tree, while a low value will create rocks instead. We will return to this subject in the actual game code, where we are using this technique extensively to produce an organic world.

3.3 Infinite World

One of our requirements is to implement an infinite game world (p. 3). This term defines that the environment is generated in real-time as the player progresses trough the world. Of course, in order to avoid performance issues, the world will also be destroyed as soon as the player leaves it, which is why it is essential to talk about the notion of consistency. If the player is coming back at a place already visited, this part of the world should be generated the same way and with all the possible changes.

Preliminary investigation

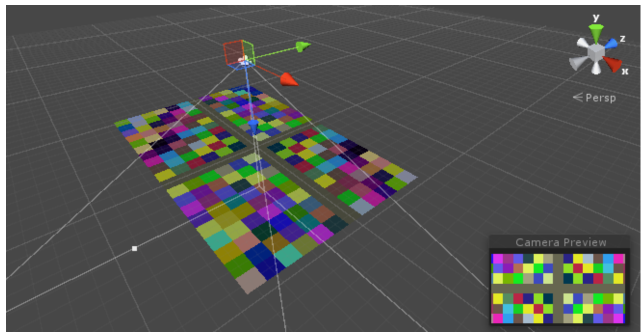


Figure 3.9

Our preliminary investigation focused on how to create consistency. We therefore created a 2D terrain made up of randomly colored cells. Each cell have a specified color depending of their position. Only the cells close to the player are drawn. When the player moves, all the cells are destroyed and new ones are constructed depending of the player position. Since the color of every cells color is determined by the 2D position, a cell with the same position will get the same color every time. In this example, the seed for the random number generator is determined by multiplying the x and z position of the cell. That means that if $x = 0$ or $z = 0$, the cell will not get any color, which can be seen on figure 3.9. The player can move from one cell to another using the arrow keys on the keyboard. When the player at some point comes back to the same the same location, the grid of colored cells will be the same. This allow this world to be consistent.

As with the colors, positions can be used as an input for the *perlin noise* algorithm. As *perlin noise* is a coherent noise algorithm, *"Passing in the same input value will always return the same output value"* [1]. When using the player position as a basis for the input, the same content will be generated at a given position in the game world and therefore making the world consistent. As we can also use *perlin noise* to distribute the objects in the world, these rules also apply to the distribution as well. By using a seed for the random number generator, when creating the permutation table, we can create different worlds or just remembering the seed if we want to create the same world. We define that seed as a *world seed* because it refers to the consistency of the world. One could simply have a world with the string *"my cool world"* as a seed, which is then translated into an integer using a hash function. This technique is used in a lot of procedural games like *Minecraft* and allows the construction of the same world with the same seed.

Mesh

In order to generate 3D content procedurally in a game, we need to have some preliminary understanding of how a 3D mesh is constructed. Meshes can be described as solid 3D objects that can vary in complexity depending on the number of vertices that the mesh is made of. We will also look into what textures is and how they are mapped in order to be used on a mesh. Furthermore, we will present how several of these meshes can be used to construct the terrain for our game world.

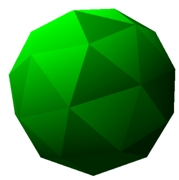


Figure 3.10: Ico sphere

Vertices and triangles

Meshes are composed of nodes called vertices and graphs known as triangles. The simplest mesh, one can create is a flat plane, which is composed of two triangles and four vertices.

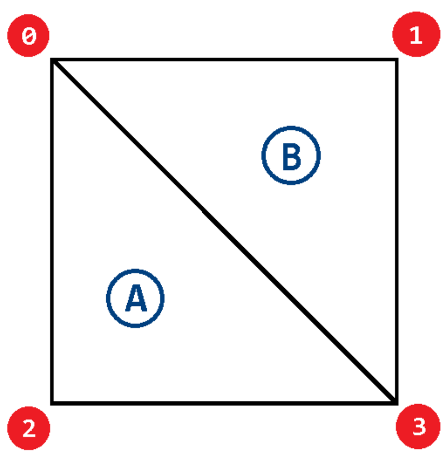


Figure 3.11: Diagram of plane

On figure 3.11 a flat plane can be seen. This plane consists of the four vertices in red from number 0 to 3 and triangle A and B. Both triangles can be represented as a directed graph, where each node in the graph is a vertex. Triangle A and B can therefore written as:

- $TriangleA = \{0, 2, 3\}$
- $TriangleB = \{0, 3, 1\}$

The construction of each triangle can either be clockwise or counter-clock wise, which determines which of the two sides are being rendered. That means that if we would like to create a cube, we should decide whether the cube should be viewed from the inside or the outside. If both sides of the plane should be rendered, we would need four triangles instead of two.

Normals

A normal is a vector perpendicular to the mesh surface. When a mesh receive light, the direction of the normal determines the brightness on its surfaces. Each vertex usually have a normal. When the brightness of the surface needs to be calculated, it is done by comparing the direction of light source and the normal [17]. If the light is coming from the same angle as the normal, the surface will be fully lit. If the light is coming from a 90° angle from that normal, the surface will not get lit. The brightness of an object therefore depends on the angle between the normal and the light source. Shown on figure 3.12

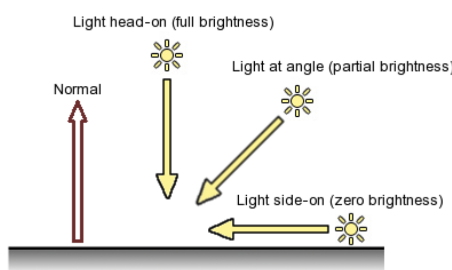


Figure 3.12: Normals[14]

Textures

After defining the vertices, triangles and normals, the mesh needs to be textured. A texture can be thought of a picture that is wrapped around the mesh in order to create *"the feel, appearance or consistency of a surface or a substance"*[5]. If one would want to create dice, one would first create a cube and then apply the texture in order to give the representation of a dice (see figure 3.13). In order to do this coordinates for the texture needs to be defined. These are known as a UV map and determines which part of the texture should be applied to each of the six faces of the cube.

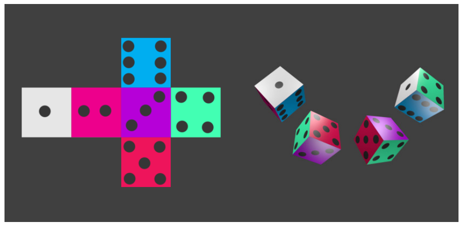


Figure 3.13: Dice UV texture on a cube[8]

When defining the UV map, we will use coordinates in two dimensions and a scale from 0 to 1 on each axis. Those coordinates are represented by U and V because x and y are already used for the vertices. We will have a UV coordinate for each vertex, which then corresponds to a certain place on the texture.

Procedural meshes for terrain modeling

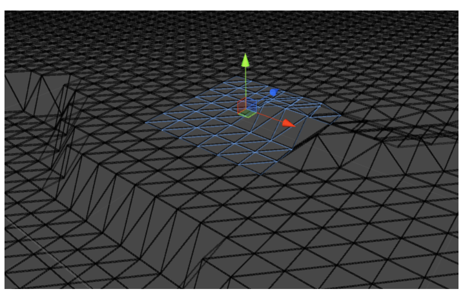


Figure 3.14: A chunk with modify height

An approach, to create the terrain for our implementation, is to build it from multiple meshes, where each of these meshes can be defined as a chunk. As we will also be explaining later in the implementation, a chunk can also contain multiple objects in the form of others meshes but for this chapter a chunk will have the definition as above. Each chunk has a specified size, which is determined by the number of vertices on each axis (see figure 3.15). One could say that the collection of chunks makes up the *game space* in the form of our *outdoor map*. (See figure 3.14)

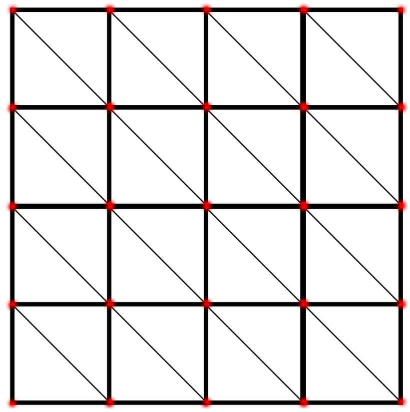


Figure 3.15: A chunk (with a chunk size of 4) : group of multiple meshes

As the world is constructed by several chunks, we can adjust the total number of vertices in the landscape by changing the number and size of the chunks, which will allow us to adjust the level of detail, draw distance in the game world. This gives us the choice to either create a large number of small chunks or a small number of larger chunks. Therefore, we will have some control over the performance and total number of vertices in the game world. Furthermore, we can utilize the illusion of the world being infinite, in the same manner as we did with the colored cells on page 25 by moving the chunks in relation to the players position.

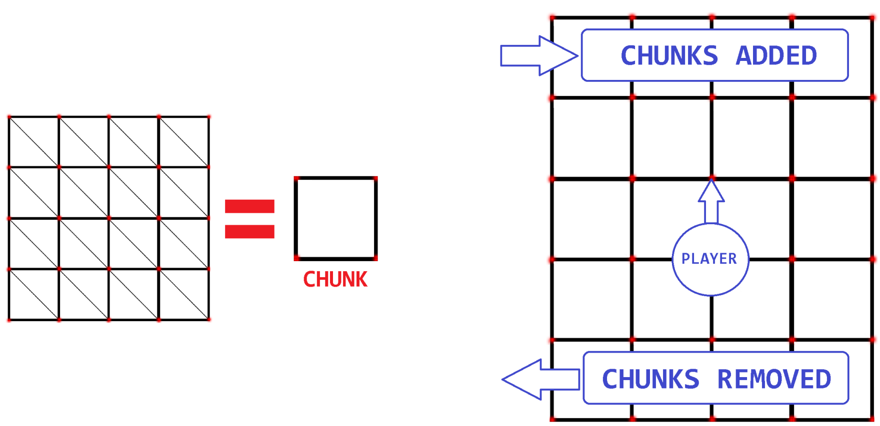


Figure 3.16: Player modify the chunks in real time

To further explain this concept, one can think of content that is being constructed and destroyed at the same time, while the player is moving in the game world. The main idea is therefore to construct the terrain in front of him and destroy the part of the terrain which is behind him. We will consequently only construct the world which is adjacent to player. It is also important that the same content is created, when the player returns to a previously visited position to keep the consistency of the world. (See figure 3.16)

When constructing or moving a chunk, it is only the height (y-axis) of the vertices that needs to be calculated in order to update the terrain. As every vertex has x and z positions in the game world, it is possible to use these, as an input for the *perlin noise* algorithm. We will then use the returned value from the *perlin noise* algorithm and apply it to the vertices y-axis, without changing the x and z position for that vertex. By using this approach we will be able to construct a landscape-like terrain where the player can walk infinitely in every direction. And as mentioned before, the permutation table can also be constructed by a random number generator using a *world seed*, which enables us to create the same world, if we use the same seed when creating the permutation table.

3.4 L-system trees

L-system (L for Lindenmayer) is a type of generative grammar and string rewriting system that can be used to produce fractals. *L-systems* was developed in 1968 by Aristid Lindenmayer, a Hungarian biologist to model and describe the behaviour of plant cells [11, preface vi].

Simple example

The main idea of this system is rather simple: One must first chose a set of symbols that can be replaced. The symbols are also called variables and is defined as V. A set of production rules is also defined that describes which symbols that can be replaced by others symbols. These rules are defined as P. Lastly, one needs to define the number of iteration n and a start symbol w, which can be a string of symbols [11, p. 4]. The *L-system* can either be stochastic or deterministic. It is stochastic if there is more than one production rule for each symbol, which also means that each rule needs a weight. This weight determines the probability for the rule to be used during the iteration. This also means that a deterministic *L-system* will always produce the same outcome. To give a basic understanding of a deterministic *L-system* we can look at the following example with the properties:

$$\begin{gathered} G = \{V, w, P\}, \\ V = \{A, B\}, \\ P = (A \rightarrow AB), (B \rightarrow BA), \\ w = A, \\ n = 3. \end{gathered}$$

The production rules $P = A \rightarrow AB$ means that A is replaced with AB and the rule $P = B \rightarrow BA$ means that B is replaced with BA. $w = A$ means that

A will be our start symbol. When the production rules and the start symbol have been defined we can iterate a number n times trough these rules in order to make a more complex string of symbols:

$$n = 0 : A,$$
$$n = 1 : AB,$$
$$n = 2 : ABBA,$$
$$n = 3 : ABBABAAB.$$

We could have as many iterations as needed, but we could also use recursion to do the iterations. With *L-systems* we can produce tree like structures by using these symbols to represent procedures. As every step can be done recursively using a divide and conquer method, each branch can be thought of a symbol. Furthermore, we can also represent the previous example in a tree-structured graph 3.17, where the depth of the graph is the number of iterations.

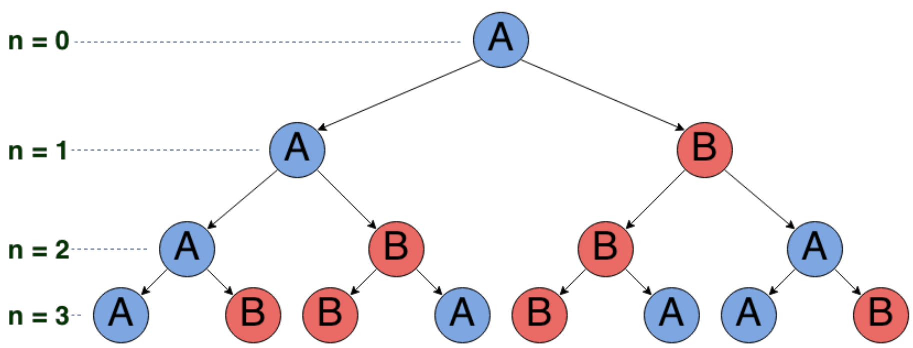


Figure 3.17: Simple L-system graph with A and B

Tree example

In order to make a tree, we will develop this example with the following properties:

$$G = \{V, C, w, P\},$$
$$V = \{A, B\},$$
$$C = \{L(, R(,)\},$$
$$P = (B \rightarrow AL(B)R(B)),$$
$$w = B,$$
$$n = 3.$$

In this case, we will add some constants, defined as C, which are symbols that cannot be replaced. This means that variables V can produce constants C but constants C cannot produce anything. After defining these properties, we will assign a procedure to each symbol. A will draw the trunk, B will draw a branch, "$L($" will rotate the following procedures by 45 degrees to the left,

"$R($" will rotate the following procedures by 45 degrees to the right and ")" will revoke the changed angle from the last "$R($" or "$P($" procedure.

$$n = 0 : B,$$
$$n = 1 : AL(B)R(B),$$
$$n = 2 : AL(AL(B)R(B))R(AL(B)R(B)),$$
$$n = 3 :$$
$$AL(AL(AL(B)R(B))R(AL(B)R(B)))R(AL(AL(B)R(B))R(AL(B)R(B)))$$

We can see that the tree gets rather complex in just a few iterations. An illustration of this above tree can be seen on figure 3.18.

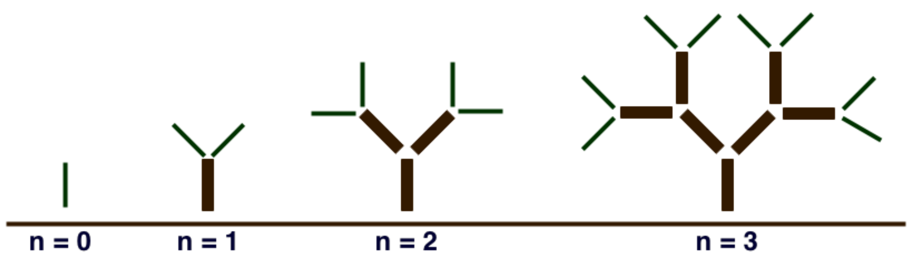


Figure 3.18: Draw tree

As mentioned before, we can have more production rules for each symbol, which means that we need to assign a weight on the graph. When combined with the previously explained concept of creating a consistent game world using a *world seed*, we can use the position or/and the height produced by the terrain modelling as a seed the the *L-system's* random number generator. This will give us the opportunity to have a deterministic and therefore consistent algorithm, while still be able to produce different trees. So even with a stochastic algorithm, we will be able to have some control over the outcome.

3.5 Storing information

In the previous sections we were talking about how to create procedural meshes and how to use it to make an infinite world. In this section, we will focus on how to store modifications in the game world. Until now, we have only focused on how to generate a somehow static and unchangeable world created using a *world seed*. As mentioned before everything in the world is based on the *world seed* and therefore produces an identical environment when the player returns to a given location. If we introduce that the player can make changes to the world, we need to store these changes in order to make a consistent world.

Storing meshes modification

There are two primary methods to store modifications in a game world. The first one is to store the final state of every object of the game. *Minecraft* uses this system. It procedurally generate a fixed sized game world from a seed and stores every object in the world that has been modified. This means that if the player would alter all block in the *Minecraft* world, we would potentially have to store a huge amount of data [18].

Storing events

The second method is to store the actual event. To give an example of this, one could think of an explosion which affect several meshes. Instead of storing each of the affected meshes, that have been altered by the explosion, one could just store the actual explosion. This means that every time that the player returns to a given position, the game will produce the same explosion, which then result in the same change to the environment. This method can be preferable in cases where a single modification affect multiple meshes.

Storing meshes modification and actions

Another possible option could be a mix between these two previous methods. If a player makes only one modification, you can store it. And after a number (depending of the system) of modification on the same meshes, it could be more efficient just to store the final state of the meshes.

Infinite storage Problem

Another issue that arises is that as the world is infinite or arbitrarily large, we could end up having to store an infinite or arbitrarily large amount of data, which is not possible. *Minecraft*, does in fact have this storage issue. To highlight this problem with numbers, we can say that at first, a *Minecraft* map size has the potential to be almost 235 petabytes. Therefore, to reduce file size and memory usage, *Minecraft's* Creator, *Markus "Notch" Persson*, decided to split the terrain into 16 x 128 x 16 chunks and store them on the user's disk[18]. In our case we need to have a limit on how much data that needs to be stored. Also, the content that needs to stored may vary depending on the nature of the content. Some things might not even have to be stored, while other more vital modification does. This can be solved, either by only storing information for a fixed time or delete stored information if the, when the player is a fixed distance from the modification. It will therefore be a question of defining the time or distance depending on all these parameters. As mentioned before, some changes are more vital than others and it is more important that the world *"feels"* consistent to the player, than the game world is actually consistent. To give an example, a player would properly not forget about destroying a building in the game world, but might not notice or remember that he accidentally destroyed a little plant.

Chapter 4

Implementation of Surogou

In this chapter we describe our final implementation of *Surogou* The purpose of developing the game is to apply procedural techniques to a game like environment and to study the issues we have set out to solve. *Surogou* is a strange procedural infinite world which the player can explore and where the procedural modelling of the landscape is done by using *perlin noise*. Objects in the world such as trees are also done procedurally but created using L-Systems and distributed with *perlin noise* as well. The terrain is made of multiple chunks that moves and updates according to the players position. *Surogou* contains no gameplay besides exploration of the infinite world and the possibility to collect coins. However, the gameplay could eventually become enriched in the future because as many possibility lies within the world. As the primary goal is to investigate procedural content generation, the focus is about the techniques used to render and create an infinite world. To further investigate the issues of storing information, we added a simple game mechanism where the player is supposed to collect coins that are distributed throughout the infinite world. We have chosen to develop the game with the *Unity Engine* and therefore we will shortly describe what Unity is and how it works in order to fully understand the structure of the game.

SUROGOU

4.1 Unity

In this section we will briefly explain what Unity is and how it operates. Unity is a system for creating multi-platform games and interactive content, where the developer uses a graphical interface for structuring the application and programming using scripts to add functionalities.

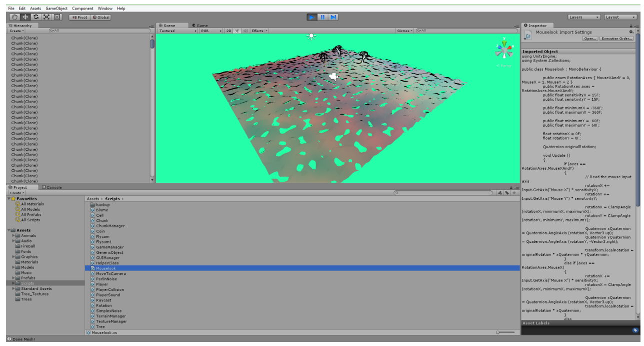


Figure 4.1: The Unity Editor

Basics

The graphical interface is known as the *Unity Editor* (figure 4.1) and is the primary tool for making games in Unity. The editor is used to create and define all the content and their properties within the game, while also creating the game environment. *Unity* works with a concept known as scenes, which are populated with *Game Objects*, which can have various behaviours and graphical representation. These *Game Objects* are the objects that make all content in the game. *Game Objects* can be extended and have different components attached to them such as meshes, scripts, sound and other graphical components. These *Game Objects* and their components can also be saved as so-called, *prefabs*, which works as a template for a game asset. A *prefab* could be a controllable player character or an instance of an AI enemy. Using *prefabs* makes it possible to change the functionality and properties for all the *Prefabs* that uses the template. Furthermore, these can be used across several projects, and their attached scripts therefore works independently. Scripting is a way to add behaviour to the game and *Unity* scripts either be written in *C#*, *UnityScript* or *Boo*. The anatomy of a basic scripts can be seen listing 4.1.

Listing 4.1: Script Anatomy

```
using UnityEngine;
using System.Collections;

public class MainPlayer : MonoBehaviour {

  // Use this for initialization
  void Start () {

  }

  // Update is called once per frame
  void Update () {

  }
}
```

The `Start()` method (line 7) is called when the game is started and is used for initialization. When the game is running the `Update()` method is called. This is done once per frame (line 12). This method could include code for movement, triggering actions, responding to player input or anything that needs to be handled during gameplay.

Creating Game Objects

Listing 4.2: Methods to create a Game Object within a script

```
1 SomeClass someClass = new SomeClass();
2 Instantiate("Some Prefab with the script attached") as SomeClass;
3 GameObject gameObject  = new GameObject().AddComponent("<SomeClass>
    ");
```

One can either place *Game Objects* in the scene by placing them in the graphical interface or instantiate them inside a script. This also means, that one would often not use the `new` keyword to create an object of a class (as seen on listing 4.2 line 1) but instead instantiate a prefab with all its scripts attached to it(line 2). Or just create a new empty *Game Object* and then attach a script to it (line 3). By using the two latter methods the *Game Object* can be seen and referenced in the graphical interface.

Referencing

As all *Game Objects* have a name, the most typically method for referencing another *Game Object* is to find it by its name as seen on listing **??** line 1. If one script component need to get access to another script component attached to the same or another *Game Object*, the `GetComponent<"otherScript">` is used as seen line 2. The last method to make a reference to another game object is to create a public *Game Object* and utilize the graphical interface to make the reference.

Listing 4.3: Reference to another game object

```
1 GameObject player = GameObject.Find("Player");
2 PlayerSound ps = player.gameObject.GetComponent<PlayerSound");
```

4.2 Process

The process of writing *Surogou* have been iterative, where several smaller programs have been created in order to investigate the different techniques. We have included some of these programs. Instructions on how to run these programs can be found on page 141.

4.3 Gameplay

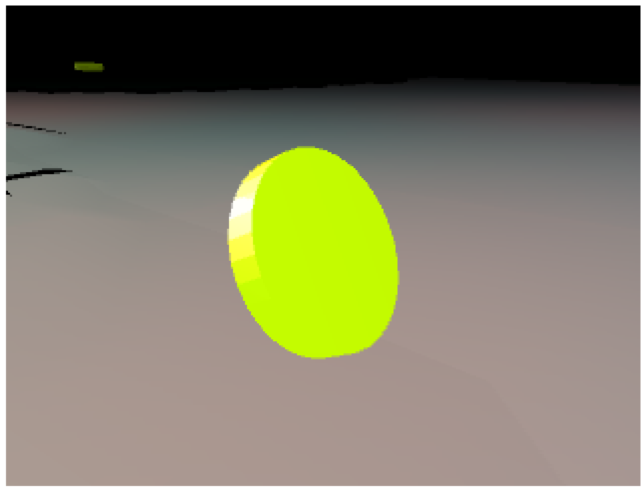


Figure 4.2: Coins to collect

The gameplay in Surogou, is primarily based on exploration, where the player is supposed collect coins. There are no way of winning the game, as the purpose of collecting coins is to create an example of how to store changes in the game world. The player uses the mouse and keyboard to walk around in the game environment, where "W" "A" "D" and "S" are used for walking and "Space" is used for jumping. When pressing the "escape" key, the player enters an ingame menu, that gives players options to quit the game or to generate a new world from a seed code.

4.4 Structure

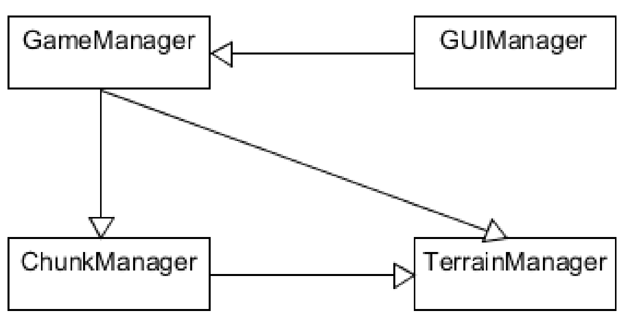


Figure 4.3: The relationship between the managers

To structure the program, we have created several main classes that acts as managers which are responsible of each part of the program. The main managers are the *Game Manager*, *Chunk Manager*, *Terrain Manager* and *GUI Manager*. Other parts of the code are independent, as each class (also known as scripts) in Unity always have a start method, which is called once when the program is executed, and an update class which is called once every frame. These classes include the code for the day/night cycle, the player's movement, sound and other independent classes. The *Game Manager* is responsible for checking in which state the program currently is in. The *GUI Manager* manages the graphical user interface. The *Chunk Manager* is responsible for the managing chunks. The *Terrain Manager* is responsible for managing the size and number of chunks, the different biome types, distribution of objects, world seed and everything related to *perlin noise*. The program can be in three different states:

1. Title menu state
2. Pause menu state
3. In-game state

In order to draw different graphical user interfaces, based on which state the program is in, we need a class to manage this. This is done in the *GUI Manager*. The *GUI manager* uses Unity's own static method `OnGUI()`, as seen on listing 4.4.

Listing 4.4: OnGUI() in the GUIManager class

```
void OnGUI(){
  if (gm.state == 0) {
    DrawTitle();
  }
  if (gm.state == 1) {
    DrawMenu();
  }
  if (gm.state == 2) {
    DrawHUD();
  }
}
```

A main game object is placed in the scene that has the four manager classes attached to it. The relation between these classes can be seen on figure 4.3. The *GUI manager* has access to the *Game Manager* in order to be able to draw different GUI depending on the state of the program. *The GUI manager* could have direct access to the *Terrain Manager* in order to change the seed directly but it seems more logically to do this trough the *Game Manager*, so the *GUI manager* is only responsible for drawing the GUI. The *Game Manager* have access to both the *Terrain Manager* and the *Chunk Manager*. It needs access to the *Terrain Manager*, as the *Terrain Manager* is responsible for parameters such as the random seed, chunk size and number of chunks. Furthermore, it needs access to the *Chunk Manager* in order to initialize and update the position of the chunks.

Figure 4.4: The title state v. 16 December 2014

The *Chunk Manager* which is responsible for managing the list of chunks in the game world also have access to the *Terrain Manager* in order to get information about chunk size and number of chunks. When the program is first executed, it enters the title menu state. This works as a setting page where different parameters can be set in order to create the world as seen on figure 4.4.

Drawing distance (number of chunk) and *world seed* can be adjusted at this page. Whenever the slider is changed, it is passed to the *Terrain Manager* trough the *Game Manager*. When clicking on the *"Generate World"*, the game goes into the in-game state (2) and the `StartGame()` method in the *Game Manager* is called as seen on listing 4.5.

Listing 4.5: Generate World in DrawTitle()

```
if (GUI.Button (new Rect (sWidthCenter, sHeightCenter+40, 100, 30),
    "Generate World")) {
    gm.state = 2;
    gm.StartGame ();
}
```

The listing 4.6 show a part of the *Game Manager* code. We need the global variables (line 6-10) that references to the `ChunkManager`, `TerrainManager` and different kind of *Game Objects* to exchange information. Furthermore, the *Game Manager* also have two variables integers that are used for storing the points and changing the state of the program. The `Start()` (line 19) method is called when the program is initialized. There are three ways of referencing other *Game Objects* and scripts and two of them are used here. As the `ChunkManager` and the `TerrainManager` are attached to the same *Game Object* as the `GameManager`, a way to reference the scripts is by using the `gameObject.GetComponent<T>()` method. `gameObject` (line 21-22) refers to the *Game Object*, which the `GameManager` script is attached to. The `T` refers to the type of object, which should be returned when calling `GetComponent<T>()`. In our case `T` is the `ChunkManager` and the `TerrainManager`. When referencing to another *Game Object*, one can also use the name of this *Game Object*. The reference is done by using the `GameObject.Find("name of a gameobject")` method on line 23-25.

Listing 4.6: The Game Manager

```
1 using UnityEngine;
2 using System.Collections;
3
4 public class GameManager : MonoBehaviour
5 {
6     private ChunkManager cm;
7     private TerrainManager tm;
8     private GameObject cam;
9     private GameObject menuCam;
10    private GameObject music;
11    private int points = 0;
12    public int state = 0; // 0 - menu, 1 - paused and 2 ingame
13
14    /**
15   * **********************
16   * called when the application is started
17   * **********************
18   **/
19    void Start ()
20    {
21        cm = gameObject.GetComponent<ChunkManager> ();
22        tm = gameObject.GetComponent<TerrainManager> ();
23        cam = GameObject.Find ("First Person Controller");
24        menuCam = GameObject.Find ("MenuCamera");
25        music = GameObject.Find ("Music");
26    }
```

When entering the menu state, the camera position is changed so it overlooks the current generated terrain instead of seeing the world trough the players eyes. This is also the reason for having two cameras instead of one. In order to swap of point of view between these two cameras, the *Game Object*, which they are attached are simply disabled or activated (line 37,40,43 and 55 to 60). The `Update()` method, which is called once per frame, first calls the `CheckInput()` method. This method (line 67-75), checks if the "escape" key has been pressed and if the program state is "in-game". If the statement returns `true`, the `menuCam` position is changed according to the players position (line 71). The next lines, takes care of executing the code, which are related to that state. Most important is the "in-game" state (`state = 2`, line 44-45), which calls the function `UpdateChunkManager()` in the `ChunkManager` and is only called if the chunks are instantiated.

Listing 4.7: The Game Manager continued

```
    /**
   * *********************
   * Core update method for the application and its different states
   * *********************
   **/
    void Update ()
    {
        CheckInput ();
        if (state == 0) {
            SetActiveObjects (false, true, true);
        }
        if (state == 1) {
            SetActiveObjects (false, true, false);
        }
        if (state == 2) {
            SetActiveObjects (true, false, false);
            if (cm.InstantiateDone) {
                cm.UpdateChunkManager ();
            }
        }
    }

    /**
   * *********************
   * Activates and deactivates cam, menucam and music gameOjects
   * *********************
   **/
    private void SetActiveObjects (bool cam, bool menuCam, bool music)
    {
        this.cam.SetActive (cam);
        this.menuCam.SetActive (menuCam);
        this.music.SetActive (music);
    }

    /**
   * *********************
   * checks the input every frame in order to see if the "escape" key was pressed, which changes the game state to 1 (pause state)
   * *********************
   **/
    private void CheckInput ()
    {
        if (state == 2) {
            if (Input.GetKeyDown (KeyCode.Escape)) {
                menuCam.transform.position = new Vector3 (cam.transform.position.x, 40, cam.transform.position.z);
                state = 1;
            }
        }
    }
```

As seen on page 40 when the GUI button labelled *"Generate World"* from the GUI manager was pressed, the `StartGame()` method was called. This method calls the `InitializeChunkManager()`, which instantiate all the chunk objects. On line 82, we also have a method for resetting the *Chunk Manager*. We use the `Reset()` method to empty the world whenever a new world needs to be created.

Listing 4.8: The Game Manager continued

```
77   /**
78  * *********************
79  * Method for resetting the chunkmanager
80  * *********************
81  **/
82   public void Reset ()
83   {
84       points = 0;
85       cm.collectedCoins.Clear();
86       cam.transform.position = new Vector3(0,4,0);
87       cm.ResetChunkManager ();
88   }
89
90   /**
91  * *********************
92  * Method to initialise the game
93  * *********************
94  **/
95   public void StartGame ()
96   {
97       cm.InitializeChunkManager ();
98   }
```

To clarify the three different states that the program can be in, we have created an activity diagram of the users actions as seen on figure 4.5.

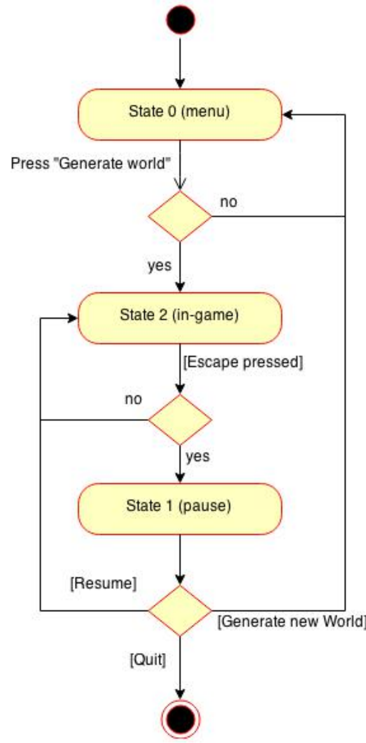


Figure 4.5: The states of the program

4.5 Rendering

The placement and update of the chunks are done in the *Chunk Manager*. A chunk consists of a procedurally generated mesh and objects related to that chunk, such as trees, rocks, fireflies and coins. As described in the last section, the chunks are initialized when the *"Generate World"* button is pressed. We have made a sequence diagram which shows the initialization (see figure 4.6. Furthermore, the `UpdateChunkManager()` is called once per frame, when the program is in the in-game state (state = 2).

Listing 4.9: The Chunk Manager

```
    public void InitializeChunkManager ()
    {
        Debug.Log ("InitializeTerrain started!");
        camPos = mainCamera.transform.position;
        tm = gameObject.GetComponent<TerrainManager> ();
        tm.CreatePerlinNoise();
        cList = new List<Chunk> ();
        r_position_x = 0;
        r_position_y = 0;
        for (int z=0; z < tm.nChunks; z++) {
```

```
39            for (int x=0; x < tm.nChunks; x++) {
40                r_position_x = (int)(x * tm.chunkSize - tm.
                      chunkSize * 0.5f * tm.nChunks + camPos.x);
41                r_position_y = (int)(z * tm.chunkSize - tm.
                      chunkSize * 0.5f * tm.nChunks + camPos.z);
42                ChunkInstance = Instantiate (chunkPrefab) as Chunk;
43                ChunkInstance.InitializeChunk (new Vector3 (
                      r_position_x, 0, r_position_y), defaultMaterial
                      , tm, this, tm.nChunks);
44                ChunkInstance.terrainGo.transform.position = new
                      Vector3 (x * tm.chunkSize - tm.chunkSize * 0.5f
                       * tm.nChunks + camPos.x,0,z * tm.chunkSize -
                      tm.chunkSize * 0.5f * tm.nChunks + camPos.z);
45                ChunkInstance.GenerateChunk ();
46                cList.Add (ChunkInstance);
47            }
48        }
49        InstantiateDone = true;
50        Name ();
51    }
```

In order to instantiate the chunks we need to get the their initial x and z position in the game world (lune 40 - 41). These positions are based by using the position of the camera, which is attached to the *Game Object* representing the player (`camPos.x` and `camPos.z`). We then create each chunk. This is done by instantiating a prefab, which have the chunk scripts attached to it. When the chunk is created, it is moved to their initial position in the game world, and lastly it added to the list `cList`, holds all the chunks in the game world. Figure 4.6 shows the initialization of the chunks.

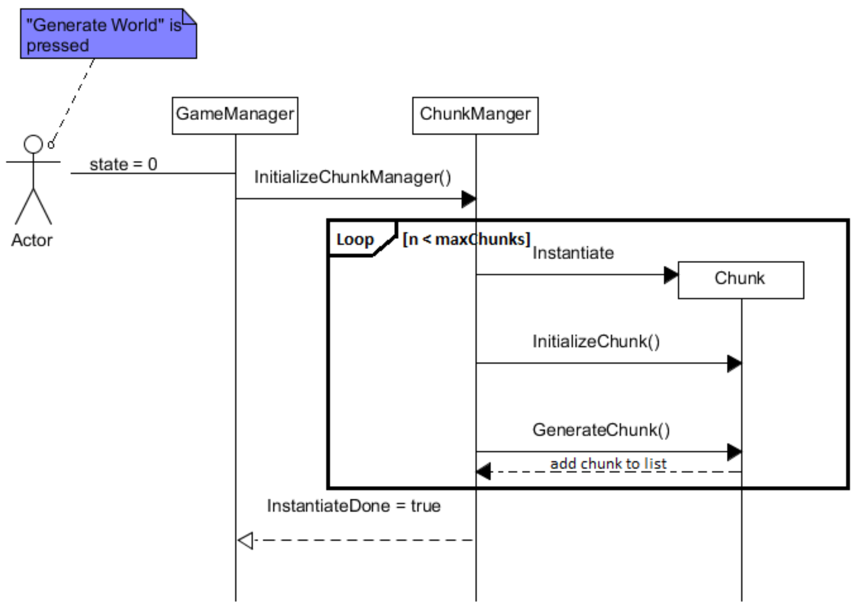


Figure 4.6: Sequence diagram of the initialization of the chunks

Listing 4.10: The Chunk Manager continued

```
    public void UpdateChunkManager ()
    {

        if (updateCount % updateFrequncy == 0) {
            StartCoroutine ("UpdateChunks", 0.0f);
        }
        camPos = mainCamera.transform.position;
        updateCount++;
    }
```

When the *Chunk Manager* is initialized (`instatiateDone`), the `UpdateChunkManager()` method is called from the *Game Manager*. In order to get a smooth frames per second we have decided to utilize coroutines, when updating the chunks. This is done in `UpdateChunks()`. Using coroutines works in the same manner as having a piece of code running in its own dedicated thread. During the update we also use two parameters called `updateFrequency`) and `yieldFactor`. The coroutine is only called when `updateCount % updateFrequency == 0`. `updateCount` is a counter which increases by one for each program cycle, whereas the `updateFrequency` defines how often the `UpdateChunks()` should be called. This means that if the `updateFrequency` is set to a value of 20 the coroutine is run at every 20 program cycle. The textttyieldFactor is used inside the coroutine. This parameter is used to give up resources by telling the program to halt for a specified amount of time by using the yield command on line 101.

Listing 4.11: The Chunk Manager continued

```
82
83     /**
84    * *********************
85    * Update Chunks
86    * *********************
87    **/
88     IEnumerator  UpdateChunks ()
89     {
90         float delta = ((tm.chunkSize) * tm.nChunks) * 0.5f;
91         if(cList.Count > 0){
92         for (int i = 0; i < tm.nChunks*tm.nChunks; i++) {
93             float dist_z = camPos.z - cList [i].terrainGo.transform
                   .localPosition.z;
94             float dist_x = camPos.x - cList [i].terrainGo.transform
                   .localPosition.x;
95
96                 if (dist_z > delta) {
97                     Vector3 newPos = new Vector3 (cList [i].
                           terrainGo.transform.localPosition.x, 0,
                           cList [i].terrainGo.transform.localPosition
                           .z + delta*2);
98                     cList [i].terrainGo.transform.position = newPos
                           ;
99                     cList [i].setPosition (newPos);
100                    cList [i].UpdateChunk ();
101                    yield return new WaitForSeconds (yieldFactor);
```

In order to move and update the chunk, as described on page 29 , we need to calculate the distance between each chunk and the current player position and store them in the two variables `dist_x` and `dist_z` (line 93 and 94). We then check if the distance is above a certain threshold by comparing the `delta` with the distance. On line 96, we can see the comparison in one of the directions. `delta` is calculated by multiplying the number of chunks and the chunk size divided by two, which represents the furthest distance from the player to the border chunks as seen on figure 4.7.

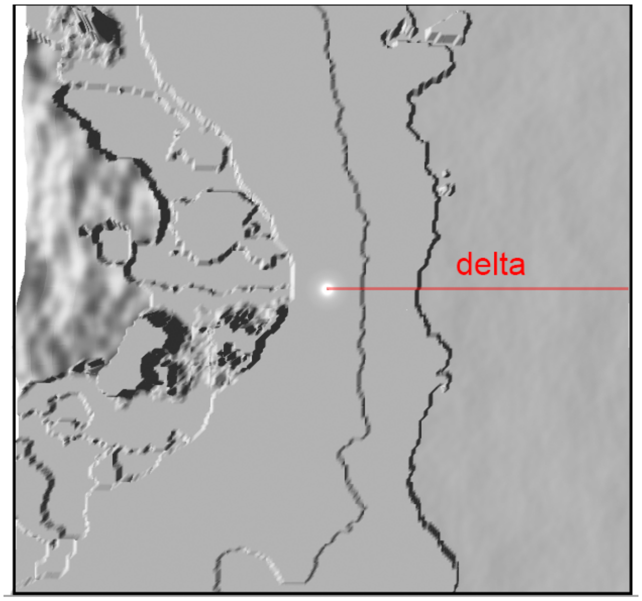


Figure 4.7: `delta` = distance from player to border

If the distance is greater than the threshold `delta` in a given direction, a new position, which is two times `delta`, is calculated and stored in the variable `newPos`(line 97). We then move the position of the `Game Object` (`terrainGO` on line 98), which have the mesh component attached to it and the chunk itself on line 99. When the chunk have been moved, we recalculate that chunk by calling the `UpdateChunk()` method (line 100). We then repeat the same steps for the three other directions.

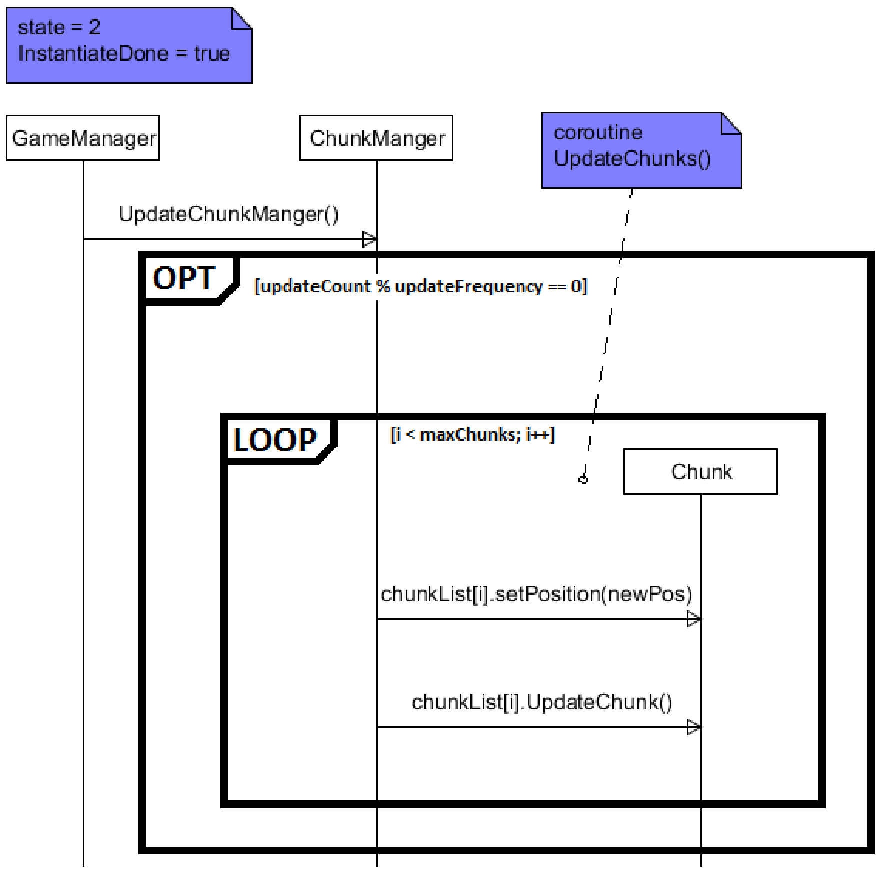


Figure 4.8: Sequence diagram of the updating the chunks

On figure 4.8 the update of chunks is shown using a sequence diagram.

4.6 Terrain

When a chunk is created, the `InitializeChunk()` method in the chunk. This works as a constructor, where variables are assigned. As previously mentioned, a chunk consists of a mesh and objects placed on that chunk. This method is only called once when the world is generated.

Listing 4.12: The Chunk

```
 /**
* *********************
* Instantiate the chunk
* *********************
**/
```

```
57     public void InitializeChunk (Vector3 position, Material
           defaultMaterial, TerrainManager terrainManager,
           ChunkManager chunkManager, int numberOfChunks)
58     {
59         terrainGo = new GameObject ("terrainMesh");
60         this.numberOfChunks = numberOfChunks;
61         this.tm = terrainManager;
62         this.cm = chunkManager;
63         this.pos = position;
64         this.defaultMaterial = defaultMaterial;
65         int numTris = terrainManager.chunkSize * terrainManager.
               chunkSize * 2;
66         vsize_x = terrainManager.chunkSize + 1;
67         vsize_z = terrainManager.chunkSize + 1;
68         int numVerts = vsize_x * vsize_z;
69 
70         // chunk objects
71         genericObjectList = new List<GenericObject> ();
72         coinList = new List<Coin> ();
73         fireFlyList = new List<FireFly> ();
74         rockList = new List<GenericObject> ();
75         treeList = new List<Tree2> ();
76 
77         vertices = new Vector3[numVerts];
78         normals = new Vector3[numVerts];
79         uv = new Vector2[numVerts];
80         triangles = new int[ numTris * 3 ];
81 
82         mesh = new Mesh ();
83         texture = new Texture2D (vsize_x, vsize_z);
84         texture.wrapMode = TextureWrapMode.Clamp;
85         texture.filterMode = FilterMode.Bilinear;
86     }
```

In order to create a mesh we need a new *Game Object* named `terrainGo` that we can use to attach the mesh data. We also need details about the number of vertices and triangles. The number of triangles (line 65) in the mesh is simply calculated by multiplying the `chunkSize` on each axis and multiplying the result by two, as a face is constructed of two triangles. The number of vertices (line 66-68) are always the `chunkSize` plus one on each axis. This might seems strange, but one can think of a plane square mesh that has the dimensions 1 x 1, which means that each axis has two vertices and not one. We also need to create a number of lists for the objects (line 71-75). We continue by creating a number of lists to hold the vertices, normals, uv and triangles (line 77-80). Moreover, we create a new `Mesh()` object (line 82). As every triangle consists of three vertices, we need to reserve space for `numTris * 3` in the triangles array (line 80). Lastly, we create the texture that can be wrapped around the mesh (line 83-85). After the mesh has been initialised, the `GenerateChunk()` is called.

Listing 4.13: The Chunk continued (GenerateChunk())

```
    /**
   * *********************
```

```
307 * Generates a new game object with a mesh attached to it
308 * **********************
309 **/
310  public void GenerateChunk ()
311  {
312      GenerateChunkData ();
313      // Create a new Mesh and populate with the data
314      mesh.vertices = vertices;
315      mesh.triangles = triangles;
316      mesh.normals = normals;
317      mesh.uv = uv;
318      mesh.RecalculateBounds ();
319      //mesh.RecalculateNormals ();
320      mesh_filter = (MeshFilter)terrainGo.AddComponent (typeof(
             MeshFilter));
321      mesh_filter.mesh = mesh;
322      mesh_collider = (MeshCollider)terrainGo.AddComponent (
             typeof(MeshCollider));
323      mesh_collider.sharedMesh = mesh;
324      mesh_renderer = (MeshRenderer)terrainGo.AddComponent (
             typeof(MeshRenderer));
325      mesh_renderer.material = defaultMaterial;
326      mesh_renderer.material.mainTexture = texture;
327      texture.Apply ();
328  }
```

On line 312 the `GenerateChunkData()` is called. This method, does not only take care of calculating the position and heights of the vertices, but also texturing and instantiation of the objects. Both heights, object distribution and texturing are done using the *perlin noise* algorithms, as we will explain later in this chapter. The calculated vertices, triangles, normals and uv are assigned to the mesh and attached to the `terrainGo` object (314-326). Lastly, we apply the texture to the mesh.

Listing 4.14: The Chunk continued (UpdateChunk())

```
 /**
* **********************
* Updates the mesh
* **********************
**/
 public void UpdateChunk ()
 {
     GenerateChunkData ();
     mesh.vertices = vertices;
     mesh.triangles = triangles;
     //mesh.RecalculateNormals ();
     mesh_collider.sharedMesh = null;
     mesh_collider.sharedMesh = mesh;
     mesh_filter.mesh = mesh;
     mesh.RecalculateBounds ();
     texture.Apply ();
 }
```

The `UpdateChunk()`, works much like `GenerateChunk()`, but this time only the calculated mesh data are assigned. Both the methods, as explain before calls the `GenerateChunkData()`.

Listing 4.15: The Chunk continued (GenerateChunkData())

```
    /**
   * **********************
   * Generate a chunk (texture, objects, vertices, normals and UV)
   * **********************
   **/
    private void GenerateChunkData ()
    {
        DestroyChunkObjects ();
        for (int z=0; z < vsize_z; z++) {
            for (int x=0; x < vsize_x; x++) {
                float posOffset_x = ((x + pos.x) / scale);
                float posOffset_z = ((z + pos.z) / scale);
                float height;
                height = tm.GetBiomes (posOffset_x, posOffset_z);
                texture.SetPixel (x, z, tm.TerrainColor (
                    posOffset_x, posOffset_z));
                PlaceObjects (posOffset_x, posOffset_z, height);
                vertices [z * vsize_x + x] = new Vector3 (x, height
                    , z);
                normals [z * vsize_x + x] = Vector3.up;
                uv [z * vsize_x + x] = new Vector2 ((float)x /
                    vsize_x, (float)z / vsize_z);
            }
        }
```

First of all, we clear the list of objects by calling the `DestroyChunkObjects()`. For every position, we get back the `height` for a position in the game world by calling the `GetBiomes()` method. The same position is used for getting the color pixel for the texture and also to see if any objects should be placed on that position. We also calculate the vertices, normals and uv for the mesh.

Listing 4.16: The Chunk continued (GenerateChunkData())

```
        for (int z=0; z < tm.chunkSize; z++) {
            for (int x=0; x < tm.chunkSize; x++) {
                int squareIndex = z * tm.chunkSize + x;
                int triOffset = squareIndex * 6;
                triangles [triOffset + 0] = z * vsize_x + x + 0;
                triangles [triOffset + 1] = z * vsize_x + x +
                    vsize_x + 0;
                triangles [triOffset + 2] = z * vsize_x + x +
                    vsize_x + 1;

                triangles [triOffset + 3] = z * vsize_x + x + 0;
                triangles [triOffset + 4] = z * vsize_x + x +
                    vsize_x + 1;
                triangles [triOffset + 5] = z * vsize_x + x + 1;
            }
        }
    }
```

Furthermore, we need to calculate the triangles for the mesh. The `triOffset` is constructed using the `squareIndex`, which allows us to populate every triangles data. Every plane is made of two triangles A and B 4.9, which is why we need to assign six vertices.

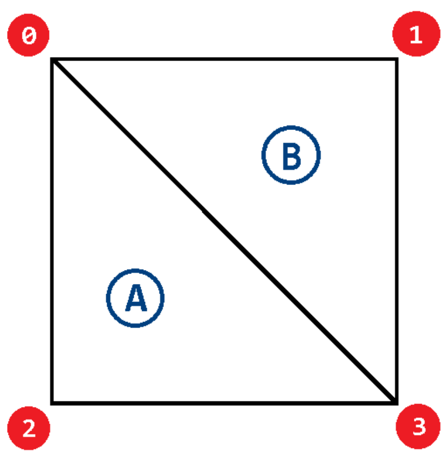


Figure 4.9: Diagram of two triangles

In every for loop, we construct two triangles. In the example on the figure 4.9, texttttriOffset will be equal to zero, `vsize_x` will be equal 2 and our coordinates z and x will be equal to 0.

- `triangles [0] = z * vsize_x + x + 0 =` **0**
- `triangles [1] = z * vsize_x + x + vsize_x + 0 =` **2**
- `triangles [2] = z * vsize_x + x + vsize_x + 1 =` **3**
- `triangles [3] = z * vsize_x + x + 0 =` **0**
- `triangles [4] = z * vsize_x + x + vsize_x + 1 =` **3**
- `triangles [5] = z * vsize_x + x + 1 =` **1**

`triangles [0]`, corresponds to node 0 (in figure 4.9), while `triangles [1]` corresponds to node 2 and so on. The triangles are constructed in a counter-clockwise order. The first triangle therefore is constructed of the nodes $0 \rightarrow 2 \rightarrow 3$ and the second triangle of the nodes $0 \rightarrow 3 \rightarrow 1$.

4.7 TerrainManager

The *Terrain Manager*, which is used extensively throughout the `GenerateChunkData()` contains all the methods, which returns data in the form of heights from the *perlin noise* algoritmes. When the game is started, the `Start()` method creates 15 *perlin noise* objects, each with a unique seed, which are used to create the permutation table in the *perlin noise*. This means, that 15 different permutation tables are created, which can be used in the *Terrain Manager*.

Listing 4.17: The Terrain Manager (Start())

```
public void CreatePerlinNoise(){
  for (int i = 0; i < 15; i++) {
    perlinNoise [i] = new PerlinNoise (seed + i);
  }
}
```

In order to illustrate how the *perlin noise* is used, we can look at the method for creating biomes, which can be seen on figure 4.10.

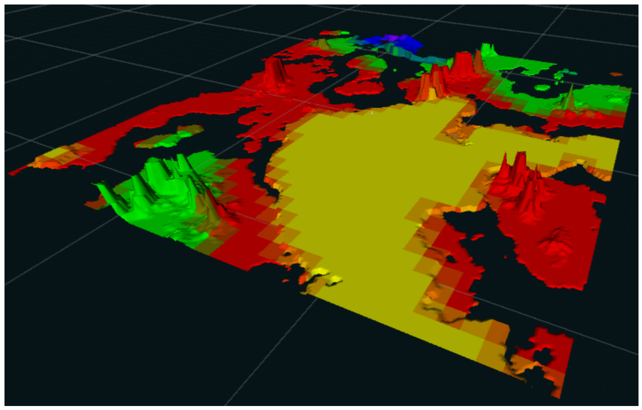


Figure 4.10: Biome types using perlin noise

The call for the method to get heights is done on line 281 in the `Chunk` scripts `GenerateChunkData()`.

Listing 4.18: The Chunk continued (GenerateChunkData())

```
281                 height = tm.GetBiomes (posOffset_x, posOffset_z);
```

As we can see this is done in the method `GetBiomes(posOffset_x,posOffset_z)` in the *Terrain Manager*. The `posOffset_x` and `posOffset_z` represent absolute coordinates in the game world, which means that the value can be positive, negative or null, depending on the players position. In listing 4.20, the code for the *GetBiomes()* can be seen.

Listing 4.19: The Terrain Manager (GetBiomes ())

```
216   public float GetBiomes (float pos_x, float pos_z)
217   {
218     float biomeNoise = perlinNoise[1].FractalNoise2D (new Vector2 (
            pos_x, pos_z), 8, 0.01f, 2f, 0.5f, 4f);
219     float elevationNoise = perlinNoise[9].FractalNoise2D(new
            Vector2(pos_x,pos_z),8,0.012f,3f,0.1f,128);
220     int type = (int)(biomeNoise) + 4;
221     if (type < 0.5f) {
222       returnType = 0;
223       biomeNoise = SeaBiome (pos_x, pos_z);
224     }
225     if (type == 1) {
226       returnType = 1;
227       biomeNoise = SeaBiome (pos_x, pos_z);
228     })
```

On line 218 the variable `biomeNoise` uses the `FractalNoise2D()` of one of the *perlin noise* objects. The arguments to control in the method are the following `Vector3 point, int octaves, float frequency, float lacunarity, float persistence, float gain`. The parameter `type` use the `biomeNoise` value in order to choose which of the 8 biome types it should use for the biomeNoise. That means that it can produce 8 different biomes types. As we want to create islands, and as the distribution of values are often closer to median value, the border values (0,1,6,7 and 8) creates a biome type, which works as a sea biome. If we would like to have more biomes, one could simply scale this number in order to return more biome types. The size of the biomes can be influencing by changing the frequency parameter. In order to create a natural transition between the different biome types, we use utilize the variable elevationNoise on line 219. This method serves as a control of the elevation of the whole world, and therefore creates a more smooth transition between the biomes. On figure 4.13 the result can be seen.

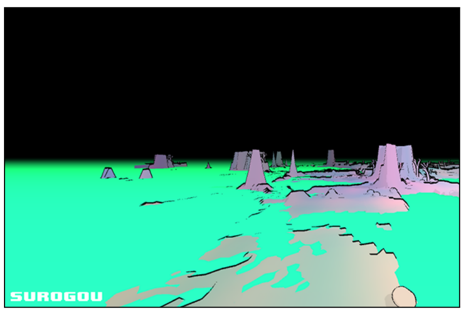


Figure 4.11: biomeNoise

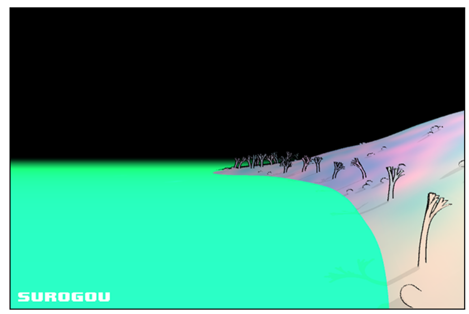


Figure 4.12: elevationNoise

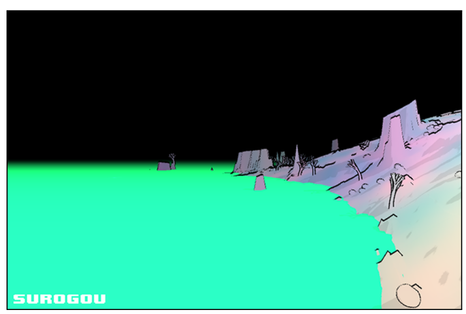


Figure 4.13: biomeNoise + elevationNoise

Listing 4.20: The Terrain Manager continued (GetBiomes ())

```
    if (type == 2) {
      returnType = 2;
      biomeNoise = HillBiome (pos_x, pos_z);
    })
```

To further give an example of how the biomes work, we will look at line 229, where the `type` parameter is equal 2, which means that the `HillBiome` method is used for retrieving height for that position. This method can be seen on listing 4.21.

Listing 4.21: The Terrain Manager (HillBiome ())

```
public float HillBiome (float pos_x, float pos_z)
{
  float noise = perlinNoise[14].FractalNoise2D (new Vector2 (
      pos_x, pos_z), 4, 0.01f, 2f, 0.5f, 15f);
  float noise2 = perlinNoise[1].FractalNoise2D (new Vector2 (
      pos_x, pos_z), 8, 0.5f, 3f, 0.23f, 3f);

  return (noise + noise2);
})
```

This `HillBiome` uses two *perlin noise* objects. `noise` one can be described as being responsible for the overall shape of the terrain, whereas `noise2` can be described as being smaller features in the terrain. This biome, illustrates how we can add several *perlin noise* algoritmes in combination in order to create more complex terrain features.

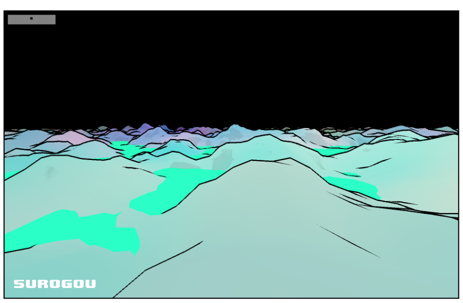


Figure 4.14: HillBiome()

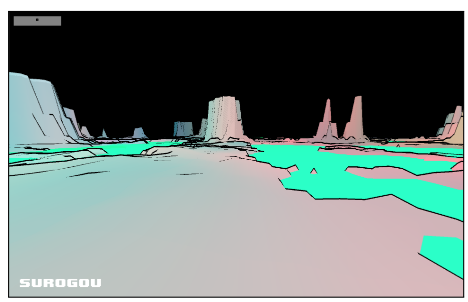


Figure 4.15: GrandBiome()

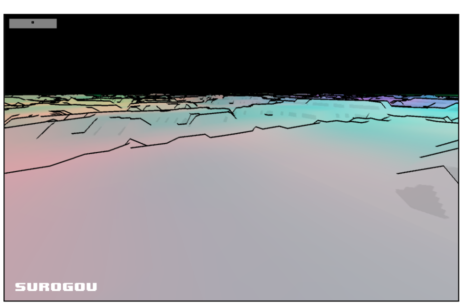


Figure 4.16: SwampBiome()

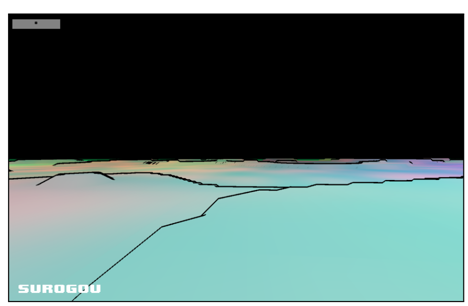


Figure 4.17: CoastBiome()

In the same manner as creating heights for the terrain, we can also use the *perlin noise* to texture the terrain while also use it to distribute object in the game world. When texturing, we use the values for colors instead of height.

4.8 Perlin Noise implementation

Our implementation of *perlin noise* is, as mentioned earlier, based on Jasper Flicks [6], who have written a tutorial on how to create a noise algorithms in the *Unity*. The algorithms is therefore implemented as described in the Terrain Modelling (p. 16) section. We have extended his implementation, to be able to create a permutation table based on the *world seed*. As mentioned before, the *world seed* is feed into the *perlin noise*, when the object is created. As mentioned earlier the permutation has 511 values ranging from and 0 - 255 values, which are then shuffled randomly in order to give the illusion of being random.

Listing 4.22: Perlin Noise (PerlinNoise())

```
const int SIZE = 511;
private int[] perm = new int[SIZE + SIZE];)
```

Listing 4.23: Perlin Noise (PerlinNoise())

```
26    public PerlinNoise (int seed)
27    {
28        UnityEngine.Random.seed = seed;
29
30        int i, j, k;
31        for (i = 0; i < SIZE; i++) {
32            // creates 0 - 255
33            perm [i] = i;
34        }
35
36        while (i > 1) {
37            i--;
38            k = perm [i];
39            j = UnityEngine.Random.Range (0, SIZE);
40            perm [i] = perm [j];
41            perm [j] = k;
42        }
43
44        for (i = 0; i < SIZE; i++) {
45            perm [SIZE + i] = perm [i];
46        }
47    })
```

On line 31 - 34 the 255 values are created for the permutation table. We then shuffle these values, using the *world seed* as a seed for the random number generator (line 38). Lastly, we copy the first half of the permutation table to the second half. This means that index 0 - 255 have the same value as index 256 - 511. This might seem strange at first, but as we need to add both x and

y position together when looking up the values in the permutation table, which can get out of bound. One way of solving this would be to do some kind of index wrapping but one could also simply solve this by doubling the size of the permutation table and copy the first half to the second.

4.9 Objects and distribution

The objects that can be created are on a chunk are the following: trees, rocks, fireflies and coins. In order to distribute objects these object in the world we can utilize the *perlin noise* algorithm. By using a threshold value, we can determine how often an object should be created in the game world. To give an example, we can see how rocks are distributed in the game world. This is done in the *PlaceObjects()* method with is a part of the`Chunk()`. The *perlin noise* used for distribution of rocks is the `RockDensity()` method in the *Terrain Manager*.

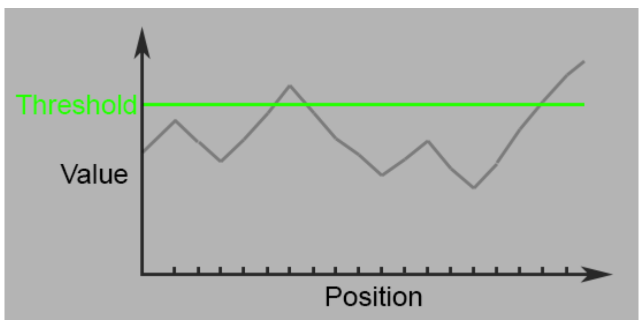


Figure 4.18: $x - axis = position$, $y - axis = value$

Listing 4.24: Chunk (PlaceObjects() (rocks))

```
            float rockDensity = tm.RockDensity (px, pz);
            if (rockDensity > 0.5f && rockList.Count < 1 && height
                > 0f && tm.GetBiomeType () == 4) {
                Random.seed = (int)(px * pz);
                float yRotation = Random.Range (0, 360);
                GenericObject rock1 = Instantiate (tm.rock, new
                    Vector3 (px * scale, (height), pz * scale),
                    Quaternion.Euler (new Vector3 (-90, yRotation,
                    0))) as GenericObject;
                rock1.transform.renderer.material.color = tm.
                    TerrainColor (px, pz);
                rockList.Add (rock1);
            })
```

The `rockDensity`, returns a float between -1 and 1. If the value is greater than *0.5f*, a rock will be created on that chunk. This value therefore represents a threshold. If we wanted to have fewer rocks, we would lower the threshold and if we would like to have more rocks, we would increase the threshold. We can also, choose which type of biome, rocks should be created by using the`GetBiomeType()`. As we can see on line 259, rocks will only be created if the biome is of the type 4, which refers to the `GrandBiome()`. We can also use more conditions such as using the height of the terrain. As we will only have rocks to appear above water level (with is 0), we check if the`height` of the terrain is greater than 0. The placement of trees, works in the same manner.

Listing 4.25: Chunk (PlaceObjects() (trees))

```
196 float treeDensity = tm.TreeDensity (px, pz);
197 if (treeDensity > 0.6f && treeList.Count < 1 && height > 1f && height < 5f && px != 0 && px != 0) {
198     Tree2 treeInstance = Instantiate (tm.tree) as Tree2;
199     int seed = (int)(px * pz);
200     Random.seed = seed;
201     float yRotation = Random.Range (0, 360);
202
203     switch (tm.GetBiomeType ())
204     {
205       case 0 : treeInstance.SetupCone (seed,20,0.0f,1.4f,10,10);
206           break;
207       case 1 : treeInstance.SetupCone (seed,15,9.0f,1.0f,2,2);
208           break;
209       case 2 : treeInstance.SetupCone (seed,7,16.0f,1.0f,8,2);
210           break;
211       case 3 : treeInstance.SetupCone (seed,15,16.0f,1.0f,10,2);
212           break;
213       case 4 : treeInstance.SetupCone (seed,15,5.0f,1.0f,3,2);
214           break;
215       case 5 : treeInstance.SetupCone (seed,15,4.0f,1.0f,5,2);
216           break;
217       case 6 : treeInstance.SetupCone (seed,12,16.0f,1.0f,1,2);
218           break;
219       default : treeInstance.SetupCone (seed,15,4.0f,1.0f,5,2);
220           break;
221     }
222
223
224     treeInstance.CreateMesh ();
225     treeInstance.renderer.material.color = tm.TerrainColor (px, pz);
226     treeInstance.plane.transform.position = new Vector3 (px * scale, (height - 1), pz * scale);
227     treeInstance.plane.transform.rotation = Quaternion.Euler (new Vector3 (0f, yRotation, 0f));
228     treeList.Add (treeInstance);
229 })
```

In order to create the same tree every time, we use the position as a seed for the random number generator in the tree algorithm, as we will explain later. Furthermore, we use the `GetBiomeType()` in order to create different types of trees in the individual biomes, as seen on line 203 - 221.

4.10 L-System implementation

As mentioned in the *L-system* trees section on page 31, we wanted to have procedural trees in the game. This proved to be a challenge which is why decided to use an existing implementation of L-System trees, which is done by *Chanfort* [3]. Therefore, we will not take the credits for the part of the game and neither explain the implementation and detail. That said, we have taken the liberty to alter the code slightly in order to make the algorithm more controllable. The original code can be found at the unity forum: *http://forum.unity3d.com/threads/l-systems-for-unity-free-script-included.272416/*

Furthermore, we have chosen to implement 6 parameters in order to add controllability to the implementation.

Listing 4.26: Controllability of the trees

```
  /***************
 * Controllabilty
**************/

  public int numberSegmentsOrigin = 15; //Number of maximum segments = number of iterations
  public float coeffAngleBranch = 4.0f; //Coeff Angle Branch
  public float coeffBranchPossibility = 1.0f; // Coeff Branch Possibility
  public int numberSegmentTrunk = 5; // Number of segment for the trunk
  public int numberSegmentFirstBranch = 2; // Number of segment before the first branch
```

- `seed` is used for the number generator in the tree algorithm, which allows to create the same tree at a specific position.
- `numberSegmentsOrigin` control the number of iteration in the algorithm. As the number of vertices grows exponentially at each iteration, it is important to keep this parameter low, as performance is greatly reduced when a lot of vertices has to be created.
- `coeffAngleBranch` controls the angles of the branches.
- `coeffBranchPossibility` controls the number of branches. By default, this coefficient is 1. If the parameter is greater, the probability to create new branches will be less.
- `numberSegmentTrunk` controls when the tree should begin to curve. If the value is low, it will begin to curve at the trunk of the tree.
- `numberSegmentFirstBranch` controls, at which iteration the first branch should be created.

Those parameters enable a large control of the L-System trees and will be used to produce different kind of tree depending on the type of their environment. This is done so the type of biome can have influence the generation of the trees in this precise area of the world.

4.11 Collection of coins

As it is mention earlier, the player can collect coins in the game world. We added this feature in order to test how, we could store changes to the world. The distribution of coins is done in the same manner as rocks and trees, so whenever a coin is collected, it should not be created again. In order to do this, we decided to store all the positions of coins that had been collected. The code for collecting coins can be seen on listing 4.27 in the script `PlayerCollision` which is attached to the *Game Object*, represented by the player.

Listing 4.27: PlayerCollision

```
public class PlayerCollision : MonoBehaviour
{
    public GameManager gm;
    private PlayerSound ps;
    Vector3 chunkPosition = new Vector3 (0, 0, 0);

    /**
   * **********************
   * Initialization
   * **********************
   **/
    void Start ()
    {
        ps = gameObject.GetComponent<PlayerSound> ();
    }

    /**
   * **********************
   * Used for collecting coins
   * **********************
   **/
    void OnControllerColliderHit (ControllerColliderHit hit)
    {
        if (hit.gameObject.name == "terrainChunk") {
            chunkPosition = hit.gameObject.transform.position;
        }

        if (hit.gameObject.name == "Coin(Clone)") {
            gm.AddPoints (1);
            gm.CollectCoin (chunkPosition);
            ps.PlayCoinSound ();
            Destroy (hit.gameObject);
        }
    }
}
```

The `OnControllerColliderHit` is called whenever the player is colliding with another *Game Object*, such as a `terrainChunk` object or a `Coin` Object. When the player collides with a `Coin` Object the position of the *Chunk*, where the coin was collected is stored. We also add a point to the player's scoreboard and destroys the coin.

Listing 4.28: The Game Manager (CollectCoin())

```
public void CollectCoin (Vector3 pos)
{
    cm.collectedCoins.Add (pos);
}
```

Listing 4.29: The Chunk Manager (collectedCoins)

```
public List<Vector3>
    collectedCoins;
```

Whenever, we want to create a coin we therefore just need to check if the coin has already been collected on that chunk. This is done in the `PlaceObjects()` method in the Chunk.

Listing 4.30: Chunk (PlaceObjects())

```
        float coinDensity = tm.CoinDensity (px, pz);

        if (coinDensity > 0.8f && coinList.Count < 1 && height
            > 0.1f && height < 4f) {
            bool alreadyCollected = false;

            for (int i = 0; i < cm.collectedCoins.Count; i++) {
                Vector3 c = cm.collectedCoins [i];
                if (c.x == terrainGo.transform.position.x && c.
                    z == terrainGo.transform.position.z) {
                    alreadyCollected = true;
                    break;
                }
            }
            if (!alreadyCollected) {
                Coin coin = tm.coin;
                coin.gameObject.name = "Coin";
                coinList.Add (Instantiate (coin, new Vector3 (
                    px * scale, (height + 1), pz * scale),
                    Quaternion.identity) as Coin);
            }
        }
```

By comparing the stored position in `collectedCoins` with the terrain object (`TerrainGo`) we can see if the coin, on that chunk, at that position have already been collected.

Chapter 5

Test and Analysis of Surogou

In this chapter we will test the requirements, first mentioned in design chapter. It will also serve as a chapter for pointing out bugs and other known issues. Performance refers to the actual performance of the *Surogou*, which is achieved by running the game on different systems and with different configurations. For the purpose of these performance tests, a special version of the *Surogou* will be used, that can record frames per second. The data from these test is then analyzed in order to give a perspective on how well the program performs and also gives a perspective on how performance can be improved in future version of the *Surogou*. We will also be testing if the game world is consistent and the controllability of our procedural methods. We will look into the consistency to see if the same game world is generated using the same *world seed*, and if coins are properly collected. To test the controllability of our procedural algorithms, we have created a special version of the game. In this version, the parameters for the algorithms can be changed instead of the draw distance slider in the normal version. The instruction to run these programs can be found in the appendix on page B. The result of this chapter is discussed further in the discussion chapter.

5.1 Performance

The performance section will be divided into four tests, where the performance is by the number of frames per second (FPS), as well as the overall stability of frames per second. The first test focus on how the program performs on different systems, whereas the second test look into how chunk sizes and the total number of chunk to be generated affects the performance. The third test focus on how much our initial efforts of optimization affects the performance. In the fourth test we investigate how much the procedural tree generation affects the performance and also compare how performance-drops relate to the number of objects created in the game world. The duration of each test is 30 seconds. We have chosen to sample at every 1/10th of a second, which makes up a total of 300 samplings for each test. An universal anomaly was discovered in the first two samples (0.0 and 0.1 seconds) in every test. Because of this, we decided not to include these samples in our average, minimum and maximum FPS tables.

First performance test

In order to get an idea of how the *Surogou* performs on different systems, we ran the game on three computers. In order to make the test comparable, we used the same configuration on each computer. The configuration for this test was as following:

`chunkSize`	6
`numberOfChunks`	24
`updateFrequency`	20
`yieldFactor`	0
vertices	20736

Figure 5.1

The amount of vertices are only related to the actual terrain and is not counting objects, such as rocks and trees.

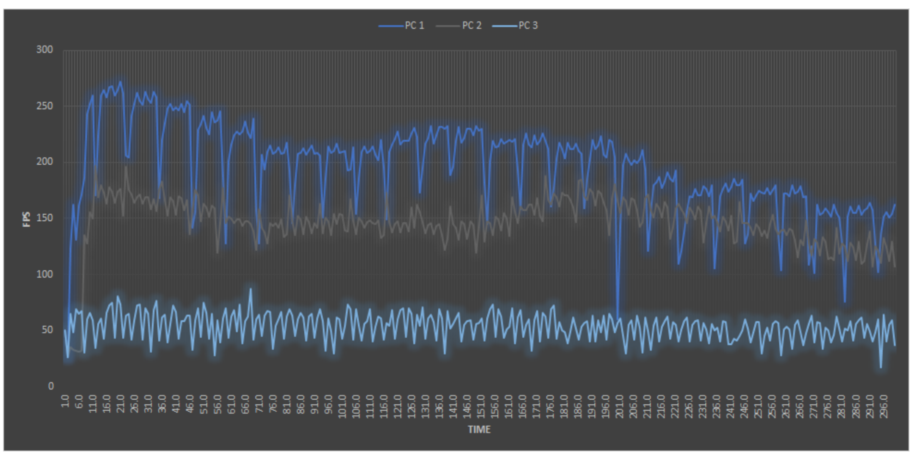


Figure 5.2

If we look at the three systems (figure 5.2), it is obvious that the hardware has a lot of influence on how the *Surogou* performs. It is self-explanatory that the drops indicate that more CPU power is needed when rendering frames at that moment, whereas the high values show little CPU computation. The dark blue graph, which is PC 1, shows the highest capability, while the CPU load is low and had an average FPS of 195. It also had the largest drops in FPS in relation to the other computers drops. The grey graph, which is *PC 2*, had an average FPS of 146, while the drops from high to low were not as significant as *PC 1*. *PC 3*, which is the light blue graph, clearly was the slowest of the three, with an average FPS of 54. PC 1 has the highest peaks and drops compared by overall performance on the three PC systems.

PC	1	2	3
Average FPS	195	146	54
Minimum FPS	60	30	17
Maximum FPS	271	196	87

Figure 5.3

Even though there is a huge difference in the lowest and highest recorded FPS at *PC 1* it was not noticeable, as the lowest FPS was high enough to give a fluent gameplay experience. PC 3 though was not performing well considering the low FPS, however the test revealed that there was no halting (lagging) during the run-through. By going through these results we chose to do performance test two, three and four on *PC 1*. Additionally we believe that *PC 1* will be the best example for showing an improvement on the performance when we compare results.

Second performance test

The purpose of this test is to see how the program performs when switching between larger and smaller chunks while also changing the amount of chunks to be generated. In order to compare the results each configuration have the same amount of vertices in the terrain. This is done by multiplying the `chunkSize` and `numberOfChunks` which always results into the same amount of vertices. When the configuration uses a high number of chunks with small size, there will be many terrain updates. When the configurations has a low number of chunks with a large chunk size there will be less terrain updates. However, the number of vertices that needs to be moved and recalculated for each chunk will be greater. We chose to run this through six configurations. The different configurations are as following:

Configuration	1	2	3	4	5	6
`chunkSize`	1	2	4	8	16	32
`numberOfChunks`	128	64	32	16	8	4

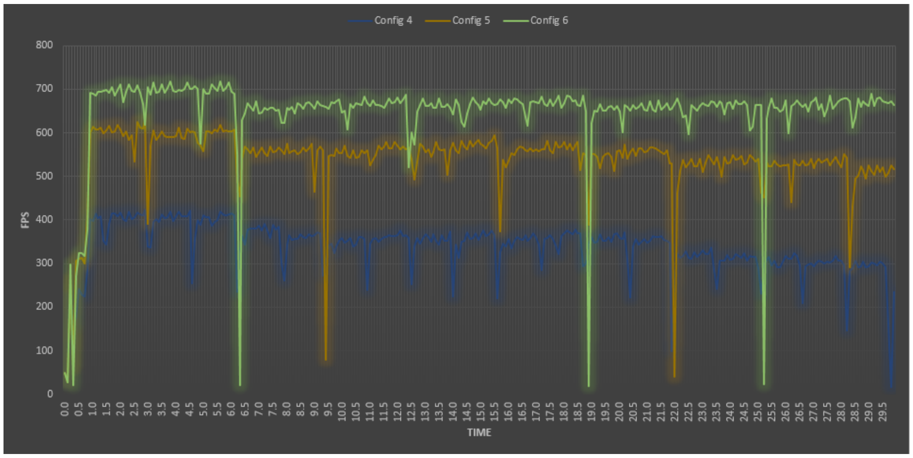
Figure 5.4

Furthermore, the `yieldFactor` was 0, while the `updateFrequency` was 20. We also decided to divide the testresults of the configurations into two graphs to avoid cluttering of the data.

Configuration	1	2	3
Average FPS	22	33	126
min FPS	15	19	16
max FPS	26	46	176

Figure 5.5

The blue graph (figure 5.5), which is *configuration 1*, shows low FPS throughout the test compared to the others and had an average of 22 FPS. We noted that the loading time was long (the initialization of the terrain) and the chunks was not moving fast enough for the camera to follow, which lead to the camera (player) moving past the game world. Additionally, we noted that objects such as trees, rocks and coins was not generated. However, the FPS remained stable with the highest reading being 26 and the lowest 15, which still lead to a less fluent experience. The yellow graph (figure 5.5), which is *configuration 2*, shows a better overall performance. During this test objects were created as opposite to *configuration 1*. The average FPS was 33, while the highest was 46 and the lowest was 19. The green graph (figure 5.5), which is *configuration 3*, shows a slight increase in performances. The lowest recorded FPS was 16 and the highest was 176, while the average FPS was 126.



Configuration	4	5	6
Average FPS	340	542	650
min FPS	16	41	18
max FPS	425	629	718

Figure 5.6

The dark blue graph (figure 5.6), shows the test with *configuration 4*. In this test the performance were further increased, while there is performance-drops which goes as low as 41 FPS. A pattern begin to appear through all the *configurations*, where the performance get higher but the drops, when compared to the average FPS, becomes increasingly significant. On the test of *configuration 6* the drops was recorded as low as 18 FPS, which makes this configuration one of the least fluent experience, with the largest drops compared to the average FPS.

It is obvious that using smaller chunks, will result in generating more chunks. If the chunks are large, more computations are needed when moving the chunks. Looking at the `chunkSize` and `numberOfChunks` between *configuration 1* and *6*, it is evident that when `chunkSize` is high and the `numberOfChunks` to be generated is low, the possibility of high performance is unlikely, if not impossible. However, when looking at the data, which can be seen in figure 5.6 and 5.5, *configuration 6* also had a biggest drops in performance. Therefore it is clear that having the *configuration* with the highest possible FPS is not necessarily the *configuration* that is most suitable for playability. Furthermore, it is also clear by looking at figure 5.6 that when `numberOfChunks` is low and `chunkSize` is high the computer takes massive performance-drops. Additionally, *configuration 4* to *6* has a hard time loading the initial objects seen by the first 0.5 seconds in the figure. It is also clear that the three most suitable *configurations* are *3*, *4* and *5*, where configuration 4 has the most stable FPS during the test, in relation to the others. This is evidence of a balanced configuration between the two parameters, where the `numberOfChunks` is 16 and the `chunkSize` is 8 (for *PC 1*). Additionally, *configuration 3* had performance issues that caused the game to halt, generate objects and then proceed, while *configuration 2* rarely created any objects within the game world. When, we conducted the tests we noted how many objects were generated. In some instances like *configuration 1* there as no objects generated at all. In *configuration 2* and *3* it started to make objects and *configuration 4* had the highst amount of objects within the game world, while *configuration 5* and *6* decreased the amount of objects that was generated. In the fourth test, we will go further into this.

After the initial test we decided to see if we could eliminate the drops, by using a higher value for the `yieldFactor`, which was initially 0. Figure 5.7 shows *configuration 4* with an altered `yieldFactor` of 2 and 4 compared to the original. The purpose of these two additional tests is to understand how the `yieldFactor` affects the performance and especially how it compares to the performance of configuration 4. Since the `yieldFactor` is a timer that delays the routine that moves chunks from the back position to the forward position, our theory was that an increase of the delay from 0 to 2 will proximately cut the number of drops in performance by half.

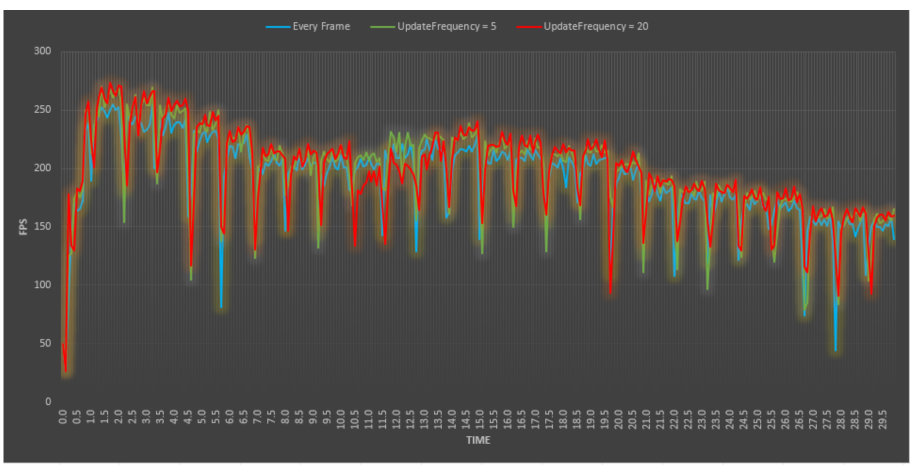
`yieldFactor`	0	2	4
Average FPS	343	348	346
min FPS	16	218	218
max FPS	434	427	435

Figure 5.7

The results of changing the `yieldFactor` is clearly improves the performance, as seen on figure 5.7. By looking at the graph, it is clear that setting the `yieldFactor` to a greater value than 0 can indeed eliminate the most of the performance drops, which happens when terrain is updated. The data in figure 5.7 shows there is a tendency to be a few differences in FPS when the `yieldFactor` changed. However, when it comes to the lowest FPS the `yieldFactor` has increased performance in terms of being more stable during the time of the test. Our reason for implementing the `yieldFactor` was originally to create a more stable performance, however we did not anticipate the amount of stability it resulted in. As it was explained earlier we expected about half of the drops in FPS to be gone, yet it seems the `yieldFactor` cuts the difference between the highest and the lowest FPS in half. It is also interesting to note that the difference from a `yieldFactor` of 2 and 4 does not affect performance by any significance.

Third performance test

During our third performance test we also focused on how to improve performance but this time the focus was on the `updateFrequency` parameter. Specifically the `updateFrequency` determines when the `UpdateChunks()` method should be called. The `UpdateChunks()` method is used for moving and recalculating chunks (terrain) in the game world. So if the `updateFrequency` is set to 5 it would mean the `UpdateChunks()` method would only be called every 5th frames. This test had the *configuration:* `chunkSize = 6`, `numberOfChunks = 24` and `yieldFactor = 0`.



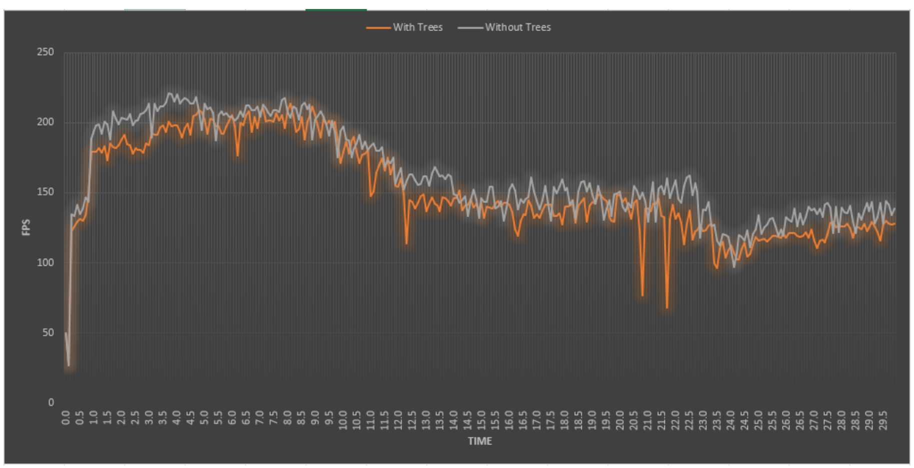
`updateFrequency`	0	5	20
Average FPS	193	200	200
min FPS	43	78	91
max FPS	255	270	273

Figure 5.8

The blue graph shows the `UpdateChunks()` when it is called at every program cycle, and have some drops in the performance. The green graph, which is the test with an `updateFrequency` of 5, shows that these drops are considerably less when compared to calling the `UpdateChunks()` at every program cycle. The last test, which is presented as the red graph, with an `updateFrequency` of 20, shows additional improvement by looking at the lowest recorded FPS which was 91. The interesting part about the results from this test is that the `updateFrequency`, much like the `yieldFactor`, increases the FPS on the lower end. From an `updateFrequency` of 0 to 5 the lowest FPS is increased from 43 to 78 FPS. Additionally from an `updateFrequency` of 5 to 20 there is an increase from 78 till 91 FPS, while the average and maximum FPS varies a little from an `updateFrequency` of 0 till 20. This indicates that the `updateFrequency` has an affect on the performance similar to the `yieldFactor`, however it is not as big.

Fourth performance test

The purpose of the forth performance test was to investigate at what degree of impact the generation of trees had on performance. The parameters for these tests were done with the following *configuration*: `chunkSize` = 4, `NumberOfchunks` = 32, `yieldFactor` = 2 and `updateFrequency` = 20.



Trees	Yes	No
Average FPS	152	163
min FPS	68	96
max FPS	213	220

Figure 5.9

On figure 5.9, the result of the fourth test can be seen. The orange graph show the test with trees, while the grey graph show the test without trees. We can notice that the performance without the trees are slightly better. Moreover, we can see some distinct drops in the graph which refers to moments when the game needs to generate a lot of trees. So we can deduce that trees got an impact on the performance of the game which can be quantified depending of the number of the trees generated. Furthermore, we will investigate if the total number of objects created in the game world got an impact on the performance. It was done by using *configuration 5* and counting the amount of objects that was created at each sampling. This shown in the following figure5.10:

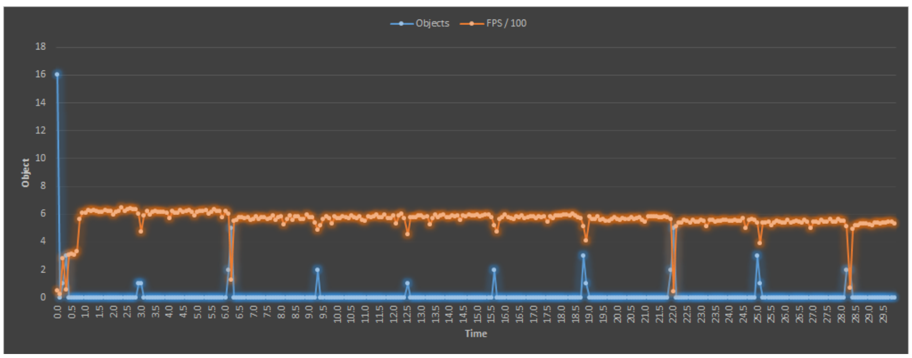


Figure 5.10

The orange graph on figure 5.10 shows the FPS divided by 100 from the test on *configuration 5* and the blue graph shows the amount of objects created on each sampling. The FPS is divided in order to see a relation between the drops in FPS and the number of objects created. The graph shows a clear relationship between the drop of FPS when a big amount of objects are being generated. It is evident that whenever there is a big drop in the performance there is a big amount of objects being generated at the same time.

Configuration	1	2	3	4	5	6
Initial	0	67	44	40	30	18
SUM 0.1-14.9	0	43	58	55	43	27
SUM 15-29.9	0	49	62	48	37	24
SUM	0	159	164	143	110	69

Figure 5.11

As it is shown in figure 5.11 there is a difference between how many objects, such as tree, rocks, coins and fireflies, are generated in each of the tests using the same *configurations*. To understand the figure better we refer to figure 4.4 which explains the mechanism of generating new objects. The initial generation is when the game world starts up, whereas the next set of sums, show how many objects were generated afterwards. While the chunks are being moved from the back to the forward positions, the objects from the back positioned chunk will be removed from the game world. This means that the sum of all objects shown in figure 5.11 is not the amount of objects at the end of the test. However, by looking at the results it is evident that the `chunkSize` is the main determinant for how many objects are created in the game world. Our reason for claiming this is that in *configuration 1* there is 128 chunks in the game world but there is no objects, while the size of each chunk is 1. Now moving from *configuration 2* till *6* the number of chunks is halved every time and the number of objects increase from *configuration 2* till *3*. However, The size of chunks do increase which indicates that the `chunkSize` parameter is a determinant for how many objects that is created. Additionally the amount of objects generated decrease through *configuration 4*, *5* and *6* which indicate that the amount objects that will be generated is also linked to the number of chunks (`numberOfChunks`) in the game world.

5.2 Consistency

In order to see if the *world seed* actually generate the same game world, we have constructed a consistency test. To do this, we implemented two test with the same *world seed*, took a screenshot for each test and compared them to see if there were any differences. In both test we used the same *world seed "my cool world"*, which should result in generating the same world. The screenshots can be seen on figure 5.12.

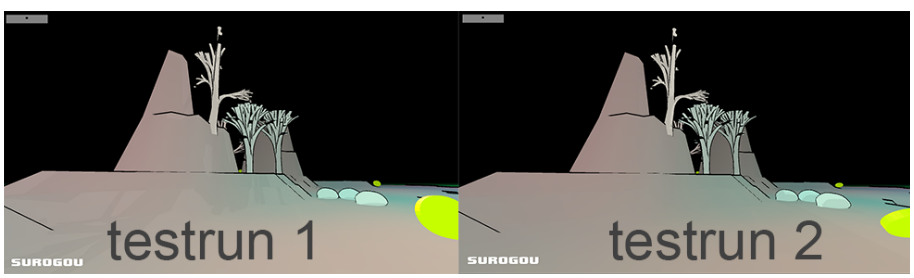


Figure 5.12

As it is seen, the two screenshots looks identical. To further prove this, we made a test which recorded the positions of each tree, coin, rock and fireflies with the two seeds *345* and *666* two times each. This resulted in two sets of two recordings where each set of recordings had identical positions of each object that was created. These tests also proved to be identical, but we have chosen not to include the data as it does not make any sense to show two sets of two recordings of identical data. We did however, observe an issue with the consistency, if we were using the same *world seed* with different chunk sizes, as seen on figure 5.13

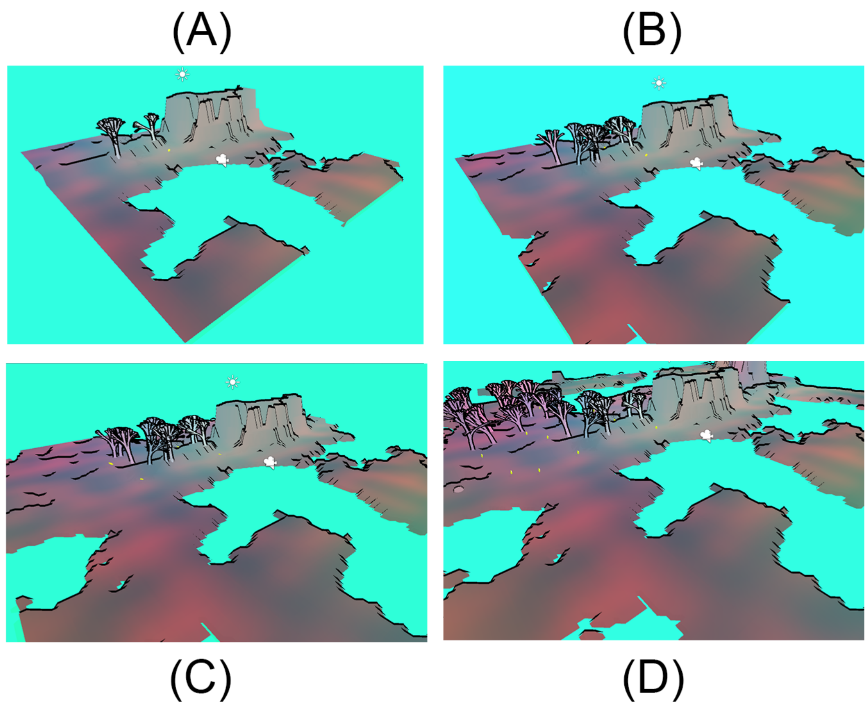


Figure 5.13

We used `chunkSize` = 4 (A), `chunkSize` = 5 (B), `chunkSize` = 6 (C) and `chunkSize` = 12 (D). By comparing screenshot A, B, C and D one can see that more trees are being generated as the `chunkSize` increases. Which shows signs of inconsistency. Our distribution of trees is done with *perlin noise* which should ensure the consistency. However, when we change the chunk size, the distribution on the chunks will also change because there is more space to distribute the objects on each chunk. Therefore, this results in an inconsistent world using the same *world seed*. Furthermore, we tested whether a collected coin would appear again if the player were to revisit the position of the collected coin. The results was that the coins were not generated again after it had been collected. We therefore concluded that the implementation of storing coins is consistent.

5.3 Controllability

The controllability does not refer to the players control, but rather the developers control of the procedural methods also explained in chapter 2 about *Degree & Dimensions of control*. From the developer's perspective, the two main procedural algorithms that was implemented should be capable of creating a wide range of different content. We tested our *perlin noise* algorithm as well as the *L-system*, that generate trees.

Perlin noise

In order to investigate how we could create different landscapes, we made a special version of the game where the user can configure the parameters of *perlin noise*.

- Octaves, is the number of iterations which creates a fractal pattern.
- Frequency, is the `frequency` of the noise.
- Lacunarity, ia a parameter which increases `frequency` at each octave.
- Persistence, is a parameter which `amplitudes` at each octave.
- Gain, which controls the maximum and minimum `amplitude`.
- Type, is either 0 (perlin noise) and 1 (value noise).

The following pictures, show how it is possible to create different content, just by changing the parameters for the coherent noise algorithm. The program *"coherent noise tester"* is attached in the appendix on page **??**.

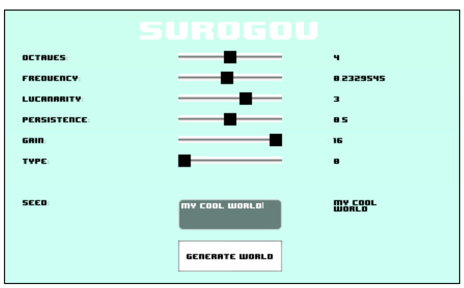


Figure 5.14: Configuration of *perlin noise*

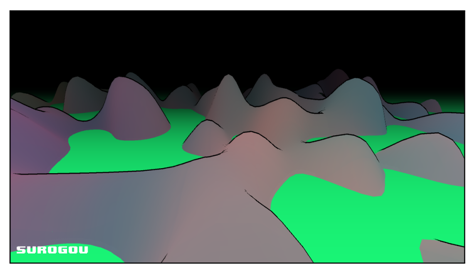


Figure 5.15

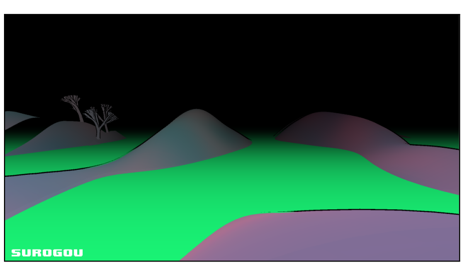


Figure 5.16

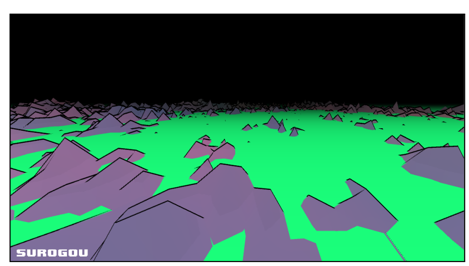


Figure 5.17

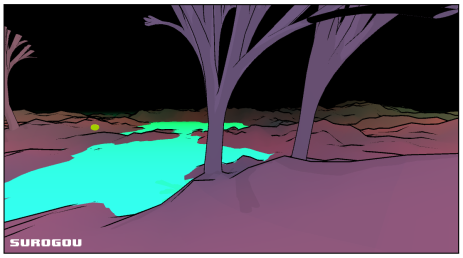


Figure 5.18

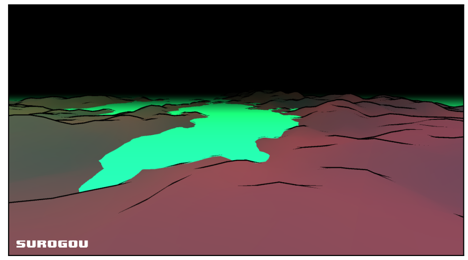


Figure 5.19

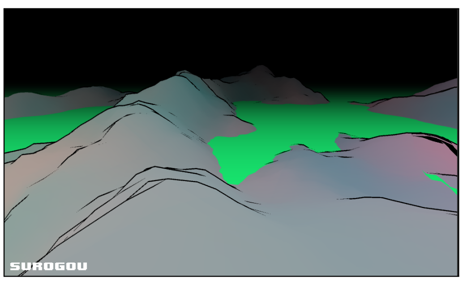


Figure 5.20

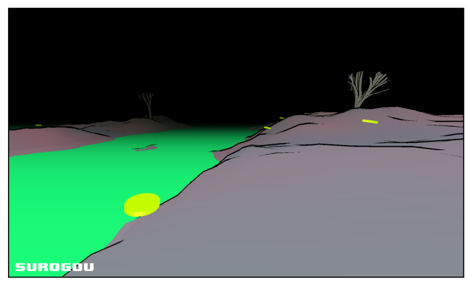


Figure 5.21

It is evident that the coherent noise are capable of creating a wide range of content.

Tree-test

The purpose of the tree-test is similar to the previous controllability test. Through this test we focus on the *degree & dimensions of control* with regards to the *L-systems* (p. 31). In order to do this, we have tested six parameters, which allow us to control how a tree should be generated (p. 62). The parameters for the tree is described on page 62. We have taken several screenshot in order to illustrate how the parameters can be used to create a wide range of trees.

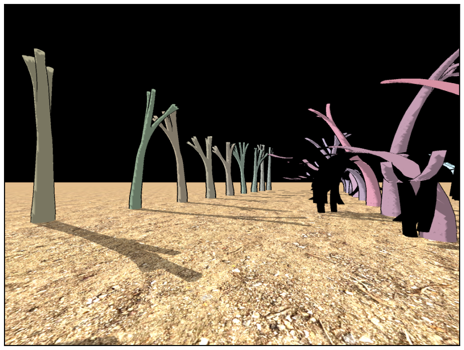


Figure 5.22

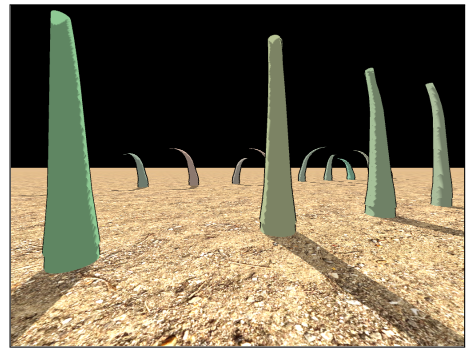


Figure 5.23

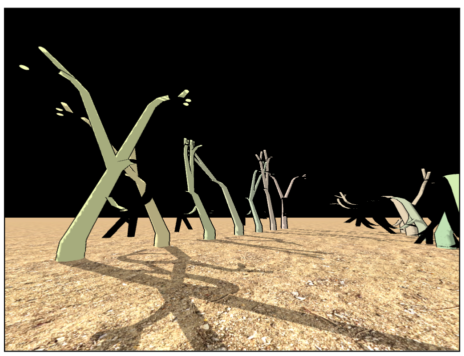


Figure 5.24

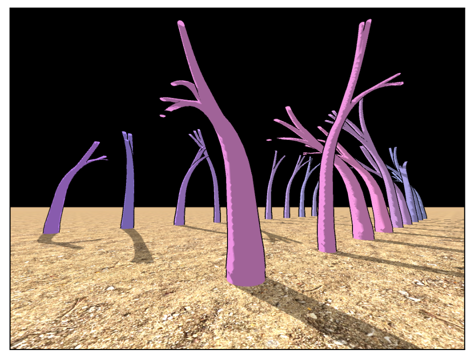


Figure 5.25

As seen in the pictures, our modification of *Chanfort* [3] L-Trees are indeed capable of a wide range of different types of trees.

5.4 Bugs and known issues

As a result of these tests, we identified a wide range of issues, which we decided to address. We managed to boost the performance by a factor of approximately 80 %, as we have made an error in *perlin noise* algoritmes, which meant that the permutation table was recalculated each time it returned a value. We also found a bug, which resulted in several identical objects being created at the same position. The reason was that a chunk could create a tree on its border vertex. This means that two trees would be generated on the same place, but on two adjacent chunks. The fix for this was to avoid objects being created on the border vertices of the mesh. This bug fix also is the reason why there were no object created, when the *chunkSize* is one, as every vertex is a border vertex. Furthermore, it is also the reason why the number of objects differs, when using different sizes of chunks. The solution to this is to have a fixed *chunkSize*, where only the number of chunks can be changed. The draw distance slider in the title screen is therefore directly linked to the number of chunks in the game world.

Chapter 6

Discussion

This chapter focuses on our testresults by discussing the data and observations in reference to our requirements Performance, Consistency, Controllability and Inifinity. Furthermore, we discuss the topics of alternative methods we could have used for updating terrain and other ways we could have stored data as well as what data that is most relevant to store.

As it is seen in the Testing Chapter 5 we have four different performance tests. In the following discussion of the performance we will our findings, specifically focus on how we can create the most optimized setup for the game.

We found that the impact on the performance had a lot to do with the number and size of the chunks. There is a fine balance between the number and sizes of the chunks. If the chunks are too large the game will halt for a moment resulting in a less fluent experience, while having many chunks, results in a low FPS due to the constant stress on the CPU. In the final version we

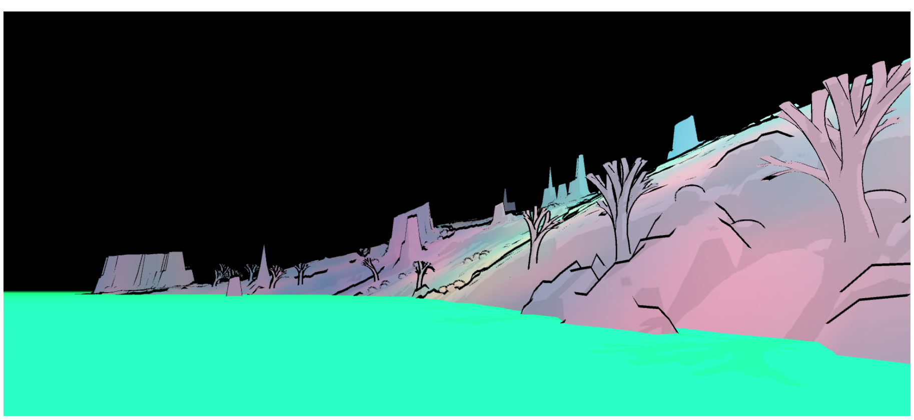


Figure 6.1: *Surogou* with a draw distance of 35

decided to set the size of the chunks to a value of 10, while the number of chunks is determined by the *"draw distance"* slider. This also eliminates the issue regarding the inconsistency of objects created when having various chunk sizes.

Furthermore, we took an initiative to increase the performance by introducing the `yieldFactor` and `updateFrequency`, as well as using coroutines. These proved to be important parameters in order to avoid performances drops when generating content in real-time. Another way to avoid performance drops was to generate all content offline instead of online. Though this would mean that we could not generate new content in real-time. One thing that could be generated offline could tree due to number of vertices that needs to be generated. Doing the instantiation of the game world we could have a list of precomputed tress which then could be cloned into the game world as needed. The results of this would be that the calculation of the vertices is not done in real-time but instead when the world is instantiated. This would also result in a longer instantiation time. In retrospective it would have been better to develop our own implementation of the tree algorithm because sufficient control of its behaviour.

Another important discussion is about the allocation of the workload between the CPU and the GPU. It was clear in the tests that lot of the performance drops was due to the increased load of the CPU, while the GPU was is only responsible for rendering the graphics. It might be beneficial to look into GPU shader programming, such as tessellation, where processing of generating the terrain could in theory be done on the GPU. The mesh collider data would though still need to be calculated on the CPU, but the whole calculation of vertices, triangles, normals and UV could be done on the GPU. This is unfortunately only possible in the professional version of Unity, where we only had access to the free version of Unity. In the same manner it could furthermore, be discussed if it could be beneficial to use other engines or write the entire engine from scratch in *C++* instead. We did some preliminary research of which engine to use when we initially decided to use the *Unity engine*, where we also looked into the newly released Unreal Engine. However, we concluded that the *Unreal Engine* was not suitable in its current state to do PCG. Furthermore, an beneficial factor of writing an engine in *C++*, would be to increase performance as we can have direct access to the CPU (assembly level) and GPU, while the disadvantage of our own engine in *C++* is that it is much more complicated and takes more time. Additionally, if one is not careful and the program is not structured well, bug fixing and memory management can become an issue.

While it is easy to prove when something is inconsistent it is harder to prove when something is consistent. We had some thoughts of how to test this but without finding a good solution as every test we thought of proved to be an identical set of data. A method to see if to signals are identical in Digital Signal Processing (DPS) is to mask them and see if they cancel each other. We thought of a way to do something similar to this but without any luck. However, explain how the consistency works within the game world by showing screenshots The only way we could figure out of how to do the test was to take screenshots, which by itself doesn't prove the consistency of the game world. As we have mentioned earlier it is more important that the player feels the game world is consistent, while the game world might not be completely consistent. An example of this the fireflies in our game world. The fireflies have small movements within a little area and when the fireflies are regenerated the position on which they moved is not accounted for during the regeneration. This is because the consistency of the fireflies are not vital to ensure the illusion of consistency within the game world.

An aspect of infinity can be discussed, as there must be a limit to infinity. This is not a limit in mathmatical terms but a limit set by the simple fact that a computer can't produce an infinite amount of possibilities. However, there can be an arbitrarily large amount of possibilities, which gives the illusion of an infinite number of possibilities. The *world seed* also have a limit as it is in fact an integer, which means that there can only be 4.294.967.295 possible worlds.

In Chapter 2, we surveyed different types of algoritmes, and while we only focused on Generative grammer (*L-systems*) and pseudo-random number generators (*perlin noise*), there are other interesting algorithms we haven't dealt with. If we had decided to go into depth with other types of algorithms, we might have discovered other interesting issues. In chapter 3.1, we looked into *value noise*, which could also have been used instead of *perlin noise*. In theory and in the literature, *perlin noise* is argued to be a better looking noise algoritmes, but in practises and by the experience we have gained from working with these two noise algoritmes, the result of the two are much alike.

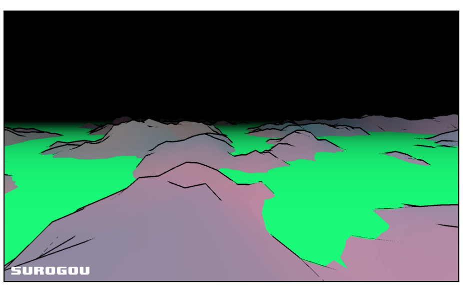


Figure 6.2: Perlin Noise

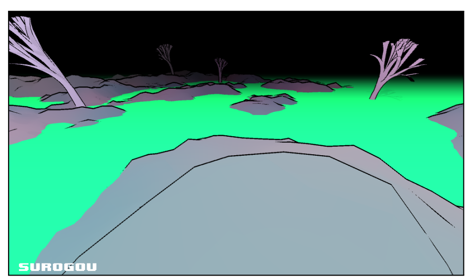


Figure 6.3: Value Noise

We also would have liked to see, how the other noise algoritmes such as *simplex noise* and *worley noise* would look, but it would require us to rewrite a large portion of the code and with the limited time of this project it was not possible.

We chose to implement coins in the game world in order to investigate the issue of storing changes. Coins are of cause not very complex game assets, but either way, implemented a simple system to store these changes. If more complex entities, such as animals were to be implemented in the game, the same principle would be used. That said, some other issues have to be addressed. The first issue is that animals are only a part of the game world, which is currently being rendered. Therefore, it is a problem to in regards of how an animal should behave, when it is not a part of the rendered game world. One way, is to use time as a factor, which means that an animal always is at the same place and doing the same thing (eating, sleeping, etc) at a given time. For this to work with our current implementation, the animal would have to be assigned to a specific chunk. This is because when the player returns to the chunk the time would be used as the parameter for the animal and this way we would be able to generated the behaviour that the animal should be doing. Another approach, that doesn't limit the movement of the animal to a

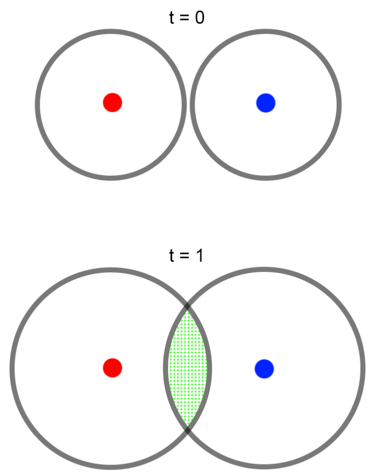


Figure 6.4: Illustration of animal behaviour. $t = time$

specified chunk is shown on figure 6.4.

In the figure 6.4 the red dot represents and an animal and the blue dot represents the player. When the player leaves the area were the animal was generated, it stores the time and position, and the actual animal is destroys like any other object game world ($t = 0$). Over time the distance (circle around the animal) is getting larger to simulate the animals movement. After some time ($t = 1$), the two circles intersect (green area), which means that there is a possibility the animal should be regenerated.

Chapter 7

Conclusion

In this project, we initially started to look into PCG as a means to improve certain aspects of future game development, involving in the increased development time due to the more content that needs to be created in videogames. We have been researching this topic, by creating our own procedurally generated game world (*Surogou*), which has given insights into issues that arises when working with PCG. We have been focusing on the topics of performance, controllability and consistency in procedurally generated game world, while also exploring the potential of generating an infinite amount of content. In the implementation of *Surogou*, we encountered several interesting issues, which are unique in nature, when working with PCG, but also unique to our implementation, while these issues and solution might also be useful in general.

We have discussed our own implementation in relation to our topics, while also discussed how we could further improve our game. As a result of this project we can conclude several aspects which makes PCG a preferable topic in videogame development for the following reasons.

The most of the development of *Surogou* has been done in less than two weeks and because the game. In contrast to the size of the game world and the variety of content, it is argued that the development time of a videogame could indeed be lessen if at least some content would be procedurally generated.

When generating content in real-time it is important not to do all calculations in one program cycle, but scatter them over several program cycles, in order to avoid performance drops. This can be done using coroutines or threads.

Through our testing of *Surogou* it has become clear that a game, which focuses on procedurally generated content can compete with a typical videogame in terms of performance. *Surogou* performs well because we have implemented parameters which lessens the work load while generating in real-time. Furthermore, there is room for additional improvements, such as allocating some of the work load in parallel threads to the GPU. Another improvement on the performance would be to ensure that objects which requires a heavy work load is only generated at the initialization of the game. In our implementation, trees are generated during real-time, which in some instances results in performance drops. These can be resolved by creating a predefine a list of unique trees during the initialization and then clone the tree into the game world when needed.

The topic of infinity was resolved by creating the world as it is consumed. This means while the player moves within the game world the content that is required will be generated at real-time.

A way to create consistency is to have a *world seed* that is used all the random number generators. This gives a game world which is deterministic while still having the generation of the game world being stochastic. This means that the same *world seed* will always generate the same game world but every unique *world seed* creates a unique game world.

Lastly the controllability is an important factor when working with procedurally generated content, as this is the only way to control the characteristics of the content that is generated.

Chapter 8

Perspectives

In this project we have investigated *content* in the form of *game bits*, *game spaces* and (to some degree) *ecosystems*. As gameplay in *Surogou* is rather limited, the next logical step would be to investigate *game systems* and *game scenarios*. These would add objective gameplay to *Surogou*. One could imagine, procedural generated quests and puzzles in the game world. Furthermore, these quest and task, could have influence on how the world is generated and would there also have an impact on *game space*. Furthermore, it could be beneficial to develop a domain-specific language for defining the *game bits* and *game spaces* (and later also *game scenarios and systems*), which would eventually eliminate the need for hard coded content, such as types of trees, objects and biome types, while also serving as a language for defining how the game should be configured. It's easy to see the potential of future versions of the game especially considering its relative short development time. That said, we had a great time working on *Surogou*.

Bibliography

[1] Jason Bevins. *What is coherent noise?*, 2005, (seen 17.12.2014). Retrieved from *http://libnoise.sourceforge.net/coherentnoise/index.html*.

[2] Chia-Jung Chan, Ruck Thawonmas, and Kuan-Ta Chen. Automatic storytelling in comics: A case study on World of Warcraft. In *Proceedings of ACM CHI 2009 (Works-in-Progress Program)*, 2009.

[3] Chanfort. *L-Tree for Unity*, 06.10.2014, (seen 12.12.2014). Retrieved from *http://u3d.as/content/chanfort/l-trees/and*.

[4] Etienne Chaudagne. *Le surpoids concerne aussi les jeux vidéo (Overweight also applied to video games) in Le Monde*, 10.10.2014, (seen 08.12.2014). Retrieved from *http://www.lemonde.fr/pixels/article/2014/10/10/le-surpoids-concerne-aussi-les-jeux-video_4504293_4408996.html*.

[5] Oxford Dictionaries. *Definition of texture*, 2014, (seen 26.11.2014). Retrieved from *http://www.oxforddictionaries.com/definition/english/texture*.

[6] Jasper Flick. *Noise, being a pseudorandom artist*, 2014, (seen 14.11.2014). Retrieved from *http://catlikecoding.com/unity/tutorials/noise/*.

[7] Mark Hendrikx, Sebastiaan Meijer, Joeri Vam Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *Delft University of Technology, the Netherlands*, 2011.

[8] Monika Karaliunaite. *3D Environment Project*, 02.04.2012, (seen 27.11.2014). Retrieved from *https://monikalt2.wordpress.com/category/3d-modelling/*.

[9] pcg.wikidot.com. *What is Procedural Content Generation?*, 2014, (seen 14.11.2014). Retrieved from *http://pcg.wikidot.com/what-pcg-is*.

[10] Ken Perlin. Improving noise. *Media Research Laboratory, Dept. of Computer Science, New York University*, 2002.

[11] Aristid Lindenmayer Przemyslaw Prusinkiewicz. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.

[12] Klaas Jan de Kraker Ruben M. Smelik and Saskia A. Groenewegen. A survey of procedural methods for terrain modelling. *Adaptive Game Content Creation using Computational Intelligence, The Hague, The Netherlands*, 2011.

[13] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research.* Springer, 2014.

[14] Unity Technologies. *Anatomy of a Mesh*, 2014, (seen 26.11.2014). Retrieved from *http://docs.unity3d.com/Manual/AnatomyofaMesh.html*.

[15] Mark Terrano. *60 FPS on Consoles*, 22.11.2014, (seen 16.12.2014). Retrieved from *http://www.giantbomb.com/60-fps-on-consoles/3015-3223/*.

[16] Julian Togelius, Emil Kastbjerg, David Schedl, and Georgios N. Yannakakis. What is procedural content generation? mario on the borderline. *Adaptive Game Content Creation using Computational Intelligence*, 2011.

[17] Max Wagner. *Generating Vertex Normals*, 12.9.2004, (seen 26.11.2014). Retrieved from *http://www.emeyex.com/site/tuts/VertexNormals.pdf*.

[18] Minecraft Wiki. *Alpha level format*, 30.11.2014, (seen 04.12.2014). Retrieved from *http://minecraft.gamepedia.com/Alpha_Level_Format*.

[19] Wikipedia. *Elite (video game)*, 12.12.2014, (seen 15.12.2014). Retrieved from *http://en.wikipedia.org/wiki/Elite_(video_game)*.

[20] Wikipedia. *The Sentinel*, 2014, (seen 13.10.2014). Retrieved from *http://en.wikipedia.org/wiki/The_Sentinel_(video_game)*.

Appendix A

Code Appendix

A.1 GUIManager.cs

Listing A.1: GUIManager.cs

```
using UnityEngine;
using System.Collections;

public class GUIManager : MonoBehaviour
{
    GameManager gm;
    public string hash = "ENTER SEED HERE";
    private int sWidthCenter;
    private int sHeightCenter;
    private float numChunksSlider = 25;
    public GUISkin gSkin;
    public Texture2D titleTexture;
    public Texture2D loadTexture;


    /**
   * *********************
   * Sets paramters on Awake
   * *********************
   **/
    void Awake ()
    {
        gm = gameObject.GetComponent<GameManager> ();
        sWidthCenter = (Screen.width / 2) - 50;
        sHeightCenter = Screen.height / 2;

    }

    /**
   * *********************
   * Draw GUI
   * *********************
   **/
    void OnGUI ()
    {
```

```
        GUI.skin = gSkin;
        if (gm.state == 0) {
            DrawTitle ();
        }
        if (gm.state == 1) {
            DrawMenu ();
        }
        if (gm.state == 2) {
            DrawHUD ();
        }
    }

    /**
   * *********************
   * Draw Title
   * *********************
   **/
    void DrawTitle ()
    {
        GUI.Box (new Rect (0, 0, Screen.width, Screen.height), "Surogou: Version 16. December 2014 \n Contact: fuad@vonloops.com \n use 'W' 'A' 'S' 'D' for movement and 'space' for jump \n \n created by: Anders Bjoern Roerbaek Pedersen, Clement Kuta & Anders Olsen \n Music and Sound by: Anders Bjoern Roerbaek Pedersen");
        /***** Background Box ******/
        GUI.Box (new Rect (0, 0, Screen.width, Screen.height), "Surogou: Version 16. December 2014");

        /***** Title ******/
        GUI.DrawTexture (new Rect (sWidthCenter - 50, (sHeightCenter - 180) - 3.5f, 200, 40), titleTexture);
        int yPosition = 100;
        GUI.Label (new Rect (sWidthCenter - 150, (sHeightCenter - yPosition) - 3.5f, 100, 20), "Draw Distance");
        numChunksSlider = GUI.HorizontalSlider (new Rect (sWidthCenter, sHeightCenter - yPosition, 100, 30), (int)numChunksSlider, 10, 35);
        int numChunksSliderInt = (int)numChunksSlider;
        GUI.Label (new Rect (sWidthCenter + 150, (sHeightCenter - yPosition) - 3.5f, 50, 20), numChunksSliderInt.ToString ());
        gm.SetNumberOfChunks ((int)numChunksSliderInt);
        yPosition = 80;
        GUI.Label (new Rect (sWidthCenter - 150, (sHeightCenter - 3.5f) - yPosition, 100, 20), "Seed:");
        hash = GUI.TextField (new Rect (sWidthCenter, sHeightCenter - yPosition, 100, 30), hash);
        GUI.Label (new Rect (sWidthCenter + 150, (sHeightCenter - 3.5f) - yPosition, 50, 20), hash);
        gm.Setseed (hash);

        /***** Generate World ******/
        if (GUI.Button (new Rect (sWidthCenter, sHeightCenter + 40, 100, 30), "Generate World")) {
            GUI.DrawTexture (new Rect (sWidthCenter, (sHeightCenter - 50) - 3.5f, 200, 40), loadTexture);
```

```
                gm.state = 2;
                gm.StartGame ();
            }
        }

        /**
       * *********************
       * Draw HUD
       * *********************
       **/
        void DrawHUD ()
        {
            GUI.DrawTexture (new Rect (10, (Screen.height - 50) - 3.5f,
                 200, 40), titleTexture);
            GUI.Box (new Rect (10, 10, 100, 20), gm.getPoints ().
                ToString ());
        }

        /**
       * *********************
       * Draw Menu
       * *********************
       **/
        void DrawMenu ()
        {
            GUI.DrawTexture (new Rect (10, (Screen.height - 50) - 3.5f,
                 200, 40), titleTexture);
        GUI.Box (new Rect (0, 0, Screen.width, Screen.height), "Surogou
            : Version 16. December 2014 \n Contact: fuad@vonloops.com \
            n use 'W' 'A' 'S' 'D' for movement and 'space' for jump \n
            \n created by: Anders Bjoern Roerbaek Pedersen, Clement
            Kuta & Anders Olsen \n Music and Sound by: Anders Bjoern
            Roerbaek Pedersen");
            if (GUI.Button (new Rect (sWidthCenter, sHeightCenter, 80,
                20), "New World")) {
                gm.Reset ();
                gm.state = 0;
            }
            if (GUI.Button (new Rect (sWidthCenter, sHeightCenter + 40,
                 80, 20), "Resume")) {
                gm.state = 2;
            }
            if (GUI.Button (new Rect (sWidthCenter, sHeightCenter + 80,
                 80, 20), "Quit")) {
                Application.Quit ();
            }
        }
}
```

A.2 GameManager.cs

Listing A.2: GameManager.cs

```
using UnityEngine;
using System.Collections;

public class GameManager : MonoBehaviour
{
    private ChunkManager cm;
    private TerrainManager tm;
    private GameObject cam;
    private GameObject menuCam;
    private GameObject music;
    private int points = 0;
    public int state = 0; // 0 - menu, 1 - paused and 2 ingame

    /**
   * *********************
   * called when the application is started
   * *********************
   **/
    void Start ()
    {
        cm = gameObject.GetComponent<ChunkManager> ();
        tm = gameObject.GetComponent<TerrainManager> ();
        cam = GameObject.Find ("First Person Controller");
        menuCam = GameObject.Find ("MenuCamera");
        music = GameObject.Find ("Music");
    }

    /**
   * *********************
   * Core update method for the application and its different states
   * *********************
   **/
    void Update ()
    {
        CheckInput ();
        if (state == 0) {
            SetActiveObjects (false, true, true);
        }
        if (state == 1) {
            SetActiveObjects (false, true, false);
        }
        if (state == 2) {
            SetActiveObjects (true, false, false);
            if (cm.InstantiateDone) {
                cm.UpdateChunkManager ();
            }
        }
    }

    /**
   * *********************
   * Activates and deactivates cam, menucam and music gameOjects
```

```
 * *********************
 **/
  private void SetActiveObjects (bool cam, bool menuCam, bool music)
  {
      this.cam.SetActive (cam);
      this.menuCam.SetActive (menuCam);
      this.music.SetActive (music);
  }

  /**
 * *********************
 * checks the input every frame in order to see if the "escape" key was pressed, which changes the game state to 1 (pause state)
 * *********************
 **/
  private void CheckInput ()
  {
      if (state == 2) {
          if (Input.GetKeyDown (KeyCode.Escape)) {
              menuCam.transform.position = new Vector3 (cam.transform.position.x, 40, cam.transform.position.z);
              state = 1;
          }
      }
  }

  /**
 * *********************
 * Method for resetting the chunkmanager
 * *********************
 **/
  public void Reset ()
  {
      points = 0;
      cm.collectedCoins.Clear();
      cam.transform.position = new Vector3(0,4,0);
      cm.ResetChunkManager ();
  }

  /**
 * *********************
 * Method to initialise the game
 * *********************
 **/
  public void StartGame ()
  {
      cm.InitializeChunkManager ();
  }

/**
 * *********************
 * Gettes and setters
 * *********************
 **/
```

```
    public void SetDebugBiomes (bool b)
    {
        tm.debugBiomes = b;
    }

    public void SetNumberOfChunks (int i)
    {
        tm.nChunks = i;
    }

    public void SetchunkSize (int i)
    {
        tm.chunkSize = i;
    }

    public void Setseed (string hash)
    {
        tm.seed = hash.GetHashCode ();
        Debug.Log(tm.seed);
    }

    public int getPoints ()
    {
        return points;
    }

    public void AddPoints (int amount)
    {
        points += amount;
    }

    public void CollectCoin (Vector3 pos)
    {
        cm.collectedCoins.Add (pos);
    }
}
```

A.3 ChunkManager.cs

Listing A.3: ChunkManager.cs

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class ChunkManager : MonoBehaviour
{
    public Material defaultMaterial;
    public Camera mainCamera;
    public Chunk chunkPrefab;
    private int updateFrequncy = 20;
    private float yieldFactor = 0.5f;
    private int updateCount = 0;
    private int r_position_x = 0;
    private int r_position_y = 0;
    private TerrainManager tm;
    [HideInInspector]
    public List<Vector3>
        collectedCoins;
    public bool InstantiateDone = false;
    private Vector3 camPos;
    private Chunk ChunkInstance;
    private List<Chunk> cList;

    /**
   * *********************
   * Initialize the chunk manager
   * *********************
   **/
    public void InitializeChunkManager ()
    {
        Debug.Log ("InitializeTerrain started!");
        camPos = mainCamera.transform.position;
        tm = gameObject.GetComponent<TerrainManager> ();
        tm.CreatePerlinNoise();
        cList = new List<Chunk> ();
        r_position_x = 0;
        r_position_y = 0;
        for (int z=0; z < tm.nChunks; z++) {
            for (int x=0; x < tm.nChunks; x++) {
                r_position_x = (int)(x * tm.chunkSize - tm.
                    chunkSize * 0.5f * tm.nChunks + camPos.x);
                r_position_y = (int)(z * tm.chunkSize - tm.
                    chunkSize * 0.5f * tm.nChunks + camPos.z);
                ChunkInstance = Instantiate (chunkPrefab) as Chunk;
                ChunkInstance.InitializeChunk (new Vector3 (
                    r_position_x, 0, r_position_y), defaultMaterial
                    , tm, this, tm.nChunks);
                ChunkInstance.terrainGo.transform.position = new
                    Vector3 (x * tm.chunkSize - tm.chunkSize * 0.5f
                     * tm.nChunks + camPos.x,0,z * tm.chunkSize -
                    tm.chunkSize * 0.5f * tm.nChunks + camPos.z);
                ChunkInstance.GenerateChunk ();
                cList.Add (ChunkInstance);
```

```
            }
        }
        InstantiateDone = true;
        Name ();
    }

    /**
   * *********************
   * Resets the chunkmanager and destroys all the objects on it
   * *********************
   **/
    public void ResetChunkManager ()
    {
        InstantiateDone = false;
        for (int i = 0; i < tm.nChunks*tm.nChunks; i++) {
            cList [i].DestroyChunkObjects ();
            Destroy (cList [i].terrainGo);
        }
        cList.Clear ();
    }

    /**
   * *********************
   * Updates the Chunk Manager
   * *********************
   **/
    public void UpdateChunkManager ()
    {

        if (updateCount % updateFrequncy == 0) {
            StartCoroutine ("UpdateChunks", 0.0f);
        }
        camPos = mainCamera.transform.position;
        updateCount++;
    }

    /**
   * *********************
   * Update Chunks
   * *********************
   **/
    IEnumerator  UpdateChunks ()
    {
        float delta = ((tm.chunkSize) * tm.nChunks) * 0.5f;
        if(cList.Count > 0){
        for (int i = 0; i < tm.nChunks*tm.nChunks; i++) {
            float dist_z = camPos.z - cList [i].terrainGo.transform
                .localPosition.z;
            float dist_x = camPos.x - cList [i].terrainGo.transform
                .localPosition.x;

                if (dist_z > delta) {
                    Vector3 newPos = new Vector3 (cList [i].
                        terrainGo.transform.localPosition.x, 0,
                        cList [i].terrainGo.transform.localPosition
                        .z + delta*2);
```

```
                    cList [i].terrainGo.transform.position = newPos
                        ;
                    cList [i].setPosition (newPos);
                    cList [i].UpdateChunk ();
                    yield return new WaitForSeconds (yieldFactor);
                }
                else if (dist_z < -delta) {
                    Vector3 newPos = new Vector3 (cList [i].
                        terrainGo.transform.localPosition.x, 0,
                        cList [i].terrainGo.transform.localPosition
                        .z - delta*2);
                    cList [i].terrainGo.transform.position = newPos
                        ;
                    cList [i].setPosition (newPos);
                    cList [i].UpdateChunk ();
                    yield return new WaitForSeconds (yieldFactor);
                }
                else if (dist_x > delta) {
                    Vector3 newPos = new Vector3 (cList [i].
                        terrainGo.transform.localPosition.x + delta
                        *2, 0, cList [i].terrainGo.transform.
                        localPosition.z);
                    cList [i].terrainGo.transform.position = newPos
                        ;
                    cList [i].setPosition (newPos);
                    cList [i].UpdateChunk ();
                    yield return new WaitForSeconds (yieldFactor);
                }
                else if (dist_x < -delta) {
                    Vector3 newPos = new Vector3 (cList [i].
                        terrainGo.transform.localPosition.x - delta
                        *2, 0, cList [i].terrainGo.transform.
                        localPosition.z);
                    cList [i].terrainGo.transform.position = newPos
                        ;
                    cList [i].setPosition (newPos);
                    cList [i].UpdateChunk ();
                    yield return new WaitForSeconds (yieldFactor);
                }
            }


        }
    }

    /**
   * *********************
   * Nameing the chunk
   * *********************
   **/
    private void Name ()
    {
        for (int i = 0; i < tm.nChunks*tm.nChunks; i++) {
            cList [i].terrainGo.transform.name = "terrainChunk";
        }
    }
}
```

A.4 TerrainManager.cs

Listing A.4: TerrainManager.cs

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class TerrainManager : MonoBehaviour
{
  public int chunkSize = 10;
  public int nChunks = 35;
  public bool debugBiomes = false;
  public Coin coin;
  public FireFly firefly;
  public GenericObject rock;
  public Tree2 tree;
  public int seed;
  private int returnType;
  private PerlinNoise[] perlinNoise = new PerlinNoise[15];

  public void CreatePerlinNoise(){
    for (int i = 0; i < 15; i++) {
      perlinNoise [i] = new PerlinNoise (seed + i);
    }
  }

  /**
   * *********************
   * returns the type of biome
   * *********************
   **/
  public int GetBiomeType ()
  {
    return returnType;
  }

  /**
   * *********************
   * Density of trees
   * *********************
   **/
  public float TreeDensity (float pos_x, float pos_z)
  {
    float noise = perlinNoise[2].FractalNoise2D (new Vector2 (pos_x
        , pos_z), 8, 2f, 1f, 1f, 1f);
    return noise;
  }

  /**
   * *********************
   * Density of coins
   * *********************
   **/
  public float CoinDensity (float pos_x, float pos_z)
  {
```

```
  float noise = perlinNoise[1].FractalNoise2D (new Vector2 (pos_x, pos_z), 8, 2f, 1f, 1f, 1f);
  return noise;
}

/**
 * **********************
 * Density of fireflies
 * **********************
 **/
public float FireflyDensity (float pos_x, float pos_z)
{
  float noise = perlinNoise[2].FractalNoise2D (new Vector2 (pos_x, pos_z), 8, 2f, 1f, 1f, 1f);
  return noise;
}

/**
 * **********************
 * Density of rocks
 * **********************
 **/
public float RockDensity (float pos_x, float pos_z)
{
  float noise = perlinNoise[3].FractalNoise2D (new Vector2 (pos_x, pos_z), 8, 0.5f, 1f, 1f, 1f);
  return noise;
}

/**
 * **********************
 * Texture the terrain
 * **********************
 **/
public Color TerrainColor (float pos_x, float pos_z)
{
  float threshold = 0.1f;
  float red = (perlinNoise[4].FractalNoise2D (new Vector2 (pos_x, pos_z), 3, 0.04f, 4f, 1f, 1f)) + 1;
  float green = (perlinNoise[5].FractalNoise2D (new Vector2 (pos_x, pos_z), 2, 0.05f, 2f, 0.75f, 1f)) + 1;
  float blue = (perlinNoise[6].FractalNoise2D (new Vector2 (pos_x, pos_z), 2, 0.03f, 2f, 0.5f, 1f)) + 1;

  if (red < threshold) {
    red = threshold;
  }
  if (green < threshold) {
    green = threshold;
  }
  if (blue < threshold) {
    blue = threshold;
  }
  return new Color (red, green, blue) * 0.5f;
}

/**
```

```
 * *********************
 * Grandbiome
 * *********************
 **/
public float GrandBiome (float pos_x, float pos_z)
{
  float noise = perlinNoise[7].FractalNoise2D (new Vector2 (pos_x
      , pos_z), 8, 0.2f, 2f, 0.5f, 3f); //
  float noise2 = perlinNoise[8].FractalNoise2D (new Vector2 (
      pos_x, pos_z), 8, 0.4f, 3f, 0.5f, 2f) - 1;
  float noise3 = perlinNoise[9].FractalNoise2D(new Vector2(pos_x,
      pos_z),8,0.001f,3f,0.1f,128);
  float[] thresh = new float[] { 1.1f, 1f, 0.9f, 0.8f, 0.7f, 0.0f
      , -1f};

  if (noise > thresh [0]) {

    noise = 16f + noise2;
  } else if (noise < thresh [0] && noise > thresh [1]) {
    noise = 8f + noise2;
  } else if (noise < thresh [1] && noise > thresh [2]) {
    noise = 3f + noise2;
  } else if (noise < thresh [2] && noise > thresh [3]) {
    noise = 2f + noise2;
  } else if (noise < thresh [3] && noise > thresh [4]) {
    noise = 1f + noise2;

  } else if (noise < thresh [4] && noise > thresh [5]) {
    noise = 0.7f + noise2;
  } else if (noise < thresh [5] && noise > thresh [6]) {
    noise = -1f + noise2;
  } else {
    noise = -1f + noise2;
  }

  returnType = 4;
  return noise + noise3;
}

/**
 * *********************
 * Swamp Biome
 * *********************
 **/
public float SwampBiome (float pos_x, float pos_z)
{
  float noise = perlinNoise[9].FractalNoise2D (new Vector2 (pos_x
      , pos_z), 8, 0.4f, 2f, 0.5f, 3f);
  float noise2 = perlinNoise[10].FractalNoise2D (new Vector2 (
      pos_x, pos_z), 8, 0.4f, 3f, 0.039f, 0.5f);
  int[] tresh = new int[]{ -3 , -2 , -1 , 0 , 1 , 2 , 3 };

  if (noise < tresh [0]) {
    noise = 5f + noise2;
  } else if (noise > tresh [0] && noise < tresh [1]) {
    noise = 4f + noise2;
  } else if (noise > tresh [1] && noise < tresh [2]) {
```

```
      noise = 3f + noise2;
    } else if (noise > tresh [2] && noise < tresh [3]) {
      noise = 2f + noise2;
    } else if (noise > tresh [3] && noise < tresh [4]) {
      noise = 1f + noise2;
    } else if (noise > tresh [4] && noise < tresh [5]) {
      noise = -1f + noise2;
    } else if (noise > tresh [5] && noise < tresh [6]) {
      noise = -1f + noise2;
    } else {
      noise = -4f + noise2;
    }
    returnType = 2;
    return noise;
  }

  /**
   * **********************
   * Coastbiome
   * **********************
   **/
  public float CoastBiome (float pos_x, float pos_z)
  {
    float noise = perlinNoise[11].FractalNoise2D (new Vector2 (
        pos_x, pos_z), 8, 0.04f, 3f, 0.2f, 16f) + 1;
    float noise2 = perlinNoise[12].FractalNoise2D (new Vector2 (
        pos_x, pos_z), 8, 0.01f, 3f, 0.5f, 16f) - 1;
    return Mathf.Abs(noise + noise2);
  }


  /**
   * **********************
   * Seabiome
   * **********************
   **/
  public float SeaBiome (float pos_x, float pos_z)
  {
    float noise = perlinNoise[13].FractalNoise2D (new Vector2 (
        pos_x, pos_z), 3, 0f, 1f, 1f, 1f);
    int[] tresh = new int[]{ -3 , -2 , -1 , 0 , 1 , 2 , 3 };
    noise = -1f;
    return noise;
  }


  /**
   * **********************
   * Hillbiome
   * **********************
   **/
  public float HillBiome (float pos_x, float pos_z)
  {
    float noise = perlinNoise[14].FractalNoise2D (new Vector2 (
        pos_x, pos_z), 4, 0.01f, 2f, 0.5f, 15f);
    float noise2 = perlinNoise[1].FractalNoise2D (new Vector2 (
        pos_x, pos_z), 8, 0.5f, 3f, 0.23f, 3f);
```

```

    return (noise + noise2);
}


/**
 * *********************
 * Biome types
 * *********************
 **/
public float GetBiomes (float pos_x, float pos_z)
{
    float biomeNoise = perlinNoise[1].FractalNoise2D (new Vector2 (
        pos_x, pos_z), 8, 0.01f, 2f, 0.5f, 4f);
    float elevationNoise = perlinNoise[9].FractalNoise2D(new
        Vector2(pos_x,pos_z),8,0.012f,3f,0.1f,128);
    int type = (int)(biomeNoise) + 4;
    if (type < 0.5f) {
      returnType = 0;
      biomeNoise = SeaBiome (pos_x, pos_z);
    }
    if (type == 1) {
      returnType = 1;
      biomeNoise = SeaBiome (pos_x, pos_z);
    }
    if (type == 2) {
      returnType = 2;
      biomeNoise = HillBiome (pos_x, pos_z);
    }
    if (type == 3) {
      returnType = 3;
      biomeNoise = SwampBiome (pos_x, pos_z);
    }
    if (type == 4) {
      returnType = 4;
      biomeNoise = GrandBiome (pos_x, pos_z);
    }
    if (type == 5) {
      returnType = 5;
      biomeNoise = CoastBiome (pos_x, pos_z);
    }
    if (type == 6) {
      returnType = 6;
      biomeNoise = SeaBiome (pos_x, pos_z);
    }
    if (type == 7) {
      returnType = 7;
      biomeNoise = SeaBiome (pos_x, pos_z);
    }
    if (type == 8) {
      returnType = 8;
      biomeNoise = SeaBiome (pos_x, pos_z);
    }
    return biomeNoise + elevationNoise
}

/**
```

```
     * *********************
     * Debug float terrain
     * *********************
     **/
    public float DebugflatTerrain ()
    {
      return 1;
    }
}
```

A.5 Chunk.cs

Listing A.5: TerrainManager.cs

```
using Unity.Engine;
using System.Collections;
using System.Collections.Generic;

[RequireComponent(typeof(MeshFilter))]
[RequireComponent(typeof(MeshRenderer))]
[RequireComponent(typeof(MeshCollider))]

public class Chunk : MonoBehaviour
{

    /***************
    * References
    **************/
    public Vector3 cameraPosition;
    private TerrainManager tm;
    private ChunkManager cm;
    public Vector3 pos;

    /***************
    * Objects
    **************/
    private List<GenericObject> genericObjectList;
    private List<Coin> coinList;
    private List<FireFly> fireFlyList;
    private List<GenericObject> rockList;
    private List<Tree2> treeList;

    /***************
    * MeshData
    **************/
    public GameObject terrainGo;
    private Vector3[] vertices = null;
    private Vector3[] normals = null;
    private Vector2[] uv = null;
    private int[] triangles;
    private int vsize_x;
    private int vsize_z;
    private int numberOfChunks;
    private Mesh mesh;
    private MeshFilter mesh_filter;
    private MeshRenderer mesh_renderer;
    private MeshCollider mesh_collider;
    const int scale = 8;

    /***************
    * Graphics
    **************/
    private Material defaultMaterial;
    private Texture2D texture;

    /**
    * *********************
```

```
* Instantiate the chunk
* *********************
**/
 public void InitializeChunk (Vector3 position, Material
     defaultMaterial, TerrainManager terrainManager,
     ChunkManager chunkManager, int numberOfChunks)
 {
     terrainGo = new GameObject ("terrainMesh");
     this.numberOfChunks = numberOfChunks;
     this.tm = terrainManager;
     this.cm = chunkManager;
     this.pos = position;
     this.defaultMaterial = defaultMaterial;
     int numTris = terrainManager.chunkSize * terrainManager.
         chunkSize * 2;
     vsize_x = terrainManager.chunkSize + 1;
     vsize_z = terrainManager.chunkSize + 1;
     int numVerts = vsize_x * vsize_z;

     // chunk objects
     genericObjectList = new List<GenericObject> ();
     coinList = new List<Coin> ();
     fireFlyList = new List<FireFly> ();
     rockList = new List<GenericObject> ();
     treeList = new List<Tree2> ();

     vertices = new Vector3[numVerts];
     normals = new Vector3[numVerts];
     uv = new Vector2[numVerts];
     triangles = new int[ numTris * 3 ];

     mesh = new Mesh ();
     texture = new Texture2D (vsize_x, vsize_z);
     texture.wrapMode = TextureWrapMode.Clamp;
     texture.filterMode = FilterMode.Bilinear;
 }

 /**
* *********************
* Method for destroying the objects on a chunk
* *********************
**/
 public void DestroyChunkObjects ()
 {
     if (genericObjectList.Count != 0) {
         for (int i = 0; i < genericObjectList.Count; i++) {
             if (genericObjectList [i] != null) {
                 genericObjectList [i].DestroySelf ();
             }
         }
         genericObjectList.Clear ();
     }

     if (coinList.Count != 0) {
         for (int i = 0; i < coinList.Count; i++) {
             if (coinList [i] != null) {
                 coinList [i].DestroySelf ();
```

```
                }
            }
            coinList.Clear ();
        }

        if (fireFlyList.Count != 0) {
            for (int i = 0; i < fireFlyList.Count; i++) {
                fireFlyList [i].DestroySelf ();
            }
            fireFlyList.Clear ();
        }

        if (rockList.Count != 0) {
            for (int i = 0; i < rockList.Count; i++) {
                rockList [i].DestroySelf ();
            }
            rockList.Clear ();
        }

        if (treeList.Count != 0) {
            for (int i = 0; i < treeList.Count; i++) {
                treeList [i].DestroySelf ();
            }
            treeList.Clear ();
        }
    }

    /**
  * *********************
  * Debug biomes, by using colored textures
  * *********************
  **/
    private void DebugBiomeColors (int x, int z)
    {
        if (tm.debugBiomes) {
            if (tm.GetBiomeType () > 0) {
                // 0 white -1
                texture.SetPixel (x, z, new Color (255, 49, 28, 1))
                    ;
                texture.Apply ();
            }
            if (tm.GetBiomeType () == 1) {
                // 1 yellow
                texture.SetPixel (x, z, new Color (255, 255, 0, 1))
                    ;
                texture.Apply ();
            }
            if (tm.GetBiomeType () == 2) {
                // 2 red
                texture.SetPixel (x, z, new Color (255, 0, 0, 1));
                texture.Apply ();
            }
            // 3 green
            if (tm.GetBiomeType () == 3) {
                texture.SetPixel (x, z, new Color (0, 255, 0, 1));
                texture.Apply ();
            }
```

```
            // 4 blue
            if (tm.GetBiomeType () == 4) {
                texture.SetPixel (x, z, new Color (0, 0, 255, 1));
                texture.Apply ();
            } // 5 violet
            if (tm.GetBiomeType () == 5) {
                texture.SetPixel (x, z, new Color (255, 0, 255, 1));
                texture.Apply ();
            } // 6 Cyan
            if (tm.GetBiomeType () == 6) {
                texture.SetPixel (x, z, new Color (0, 255, 255, 1));
                texture.Apply ();
            }
            if (tm.GetBiomeType () > 7) {
                texture.SetPixel (x, z, new Color (10, 150, 130, 1));
                texture.Apply ();
            }
        }
    }

    /**
   * *********************
   * Instantate and placement of objects on a chunk
   * *********************
   **/
    private void PlaceObjects (float px, float pz, float height)
    {
        bool isEdge = false;
        if (pz * scale % tm.chunkSize == 0 || px * scale % tm.chunkSize == 0) {
            isEdge = true;
        }
        if (!isEdge) {
            /* ****************************************************************************************
            */
            float treeDensity = tm.TreeDensity (px, pz);
            if (treeDensity > 0.6f && treeList.Count < 1 && height > 1f && height < 5f && px != 0 && px != 0) {
                Tree2 treeInstance = Instantiate (tm.tree) as Tree2;
                int seed = (int)(px * pz);
                Random.seed = seed;
                float yRotation = Random.Range (0, 360);

                switch (tm.GetBiomeType ())
                {
                  case 0 : treeInstance.SetupCone (seed,20,0.0f,1.4f,10,10);
                      break;
                  case 1 : treeInstance.SetupCone (seed,15,9.0f,1.0f,2,2);
                      break;
```

```
                    case 2 : treeInstance.SetupCone (seed,7,16.0f,1.0f,8,2);
                        break;
                    case 3 : treeInstance.SetupCone (seed,15,16.0f,1.0f,10,2);
                        break;
                    case 4 : treeInstance.SetupCone (seed,15,5.0f,1.0f,3,2);
                        break;
                    case 5 : treeInstance.SetupCone (seed,15,4.0f,1.0f,5,2);
                        break;
                    case 6 : treeInstance.SetupCone (seed,12,16.0f,1.0f,1,2);
                        break;
                    default : treeInstance.SetupCone (seed,15,4.0f,1.0f,5,2);
                        break;
                }


                treeInstance.CreateMesh ();
                treeInstance.renderer.material.color = tm.TerrainColor (px, pz);
                treeInstance.plane.transform.position = new Vector3 (px * scale, (height - 1), pz * scale);
                treeInstance.plane.transform.rotation = Quaternion.Euler (new Vector3 (0f, yRotation, 0f));
                treeList.Add (treeInstance);
            }

            /* ************************************************************************************************ */
            float fireflyDensity = tm.FireflyDensity (px, pz);
            if (fireflyDensity > 0.8f && fireFlyList.Count < 1 && height < 1f) {
                fireFlyList.Add (Instantiate (tm.firefly, new Vector3 (px * scale, (height + 10), pz * scale), Quaternion.identity) as FireFly);
            }

            /* ************************************************************************************************ */
            float coinDensity = tm.CoinDensity (px, pz);

            if (coinDensity > 0.8f && coinList.Count < 1 && height > 0.1f && height < 4f) {
                bool alreadyCollected = false;

                for (int i = 0; i < cm.collectedCoins.Count; i++) {
                    Vector3 c = cm.collectedCoins [i];
                    if (c.x == terrainGo.transform.position.x && c.z == terrainGo.transform.position.z) {
                        alreadyCollected = true;
                        break;
```

```
                    }
                }
                if (!alreadyCollected) {
                    Coin coin = tm.coin;
                    coin.gameObject.name = "Coin";
                    coinList.Add (Instantiate (coin, new Vector3 (
                        px * scale, (height + 1), pz * scale),
                        Quaternion.identity) as Coin);
                }
            }

            /*
                ****************************************************************************************
                */
            float rockDensity = tm.RockDensity (px, pz);
            if (rockDensity > 0.5f && rockList.Count < 1 && height
                > 0f && tm.GetBiomeType () == 4) {
                Random.seed = (int)(px * pz);
                float yRotation = Random.Range (0, 360);
                GenericObject rock1 = Instantiate (tm.rock, new
                    Vector3 (px * scale, (height), pz * scale),
                    Quaternion.Euler (new Vector3 (-90, yRotation,
                    0))) as GenericObject;
                rock1.transform.renderer.material.color = tm.
                    TerrainColor (px, pz);
                rockList.Add (rock1);
            }
        }
    }
    /**
   * **********************
   * Generate a chunk (texture, objects, vertices, normals and UV)
   * **********************
   **/
    private void GenerateChunkData ()
    {
        DestroyChunkObjects ();
        for (int z=0; z < vsize_z; z++) {
            for (int x=0; x < vsize_x; x++) {
                float posOffset_x = ((x + pos.x) / scale);
                float posOffset_z = ((z + pos.z) / scale);
                float height;
                height = tm.GetBiomes (posOffset_x, posOffset_z);
                texture.SetPixel (x, z, tm.TerrainColor (
                    posOffset_x, posOffset_z));
                PlaceObjects (posOffset_x, posOffset_z, height);
                vertices [z * vsize_x + x] = new Vector3 (x, height
                    , z);
                normals [z * vsize_x + x] = Vector3.up;
                uv [z * vsize_x + x] = new Vector2 ((float)x /
                    vsize_x, (float)z / vsize_z);
            }
        }

        for (int z=0; z < tm.chunkSize; z++) {
            for (int x=0; x < tm.chunkSize; x++) {
                int squareIndex = z * tm.chunkSize + x;
```

```
                int triOffset = squareIndex * 6;
                triangles [triOffset + 0] = z * vsize_x + x + 0;
                triangles [triOffset + 1] = z * vsize_x + x + vsize_x + 0;
                triangles [triOffset + 2] = z * vsize_x + x + vsize_x + 1;

                triangles [triOffset + 3] = z * vsize_x + x + 0;
                triangles [triOffset + 4] = z * vsize_x + x + vsize_x + 1;
                triangles [triOffset + 5] = z * vsize_x + x + 1;
            }
        }
    }

    /**
   * *********************
   * Generates a new game object with a mesh attached to it
   * *********************
   **/
    public void GenerateChunk ()
    {
        GenerateChunkData ();
        // Create a new Mesh and populate with the data
        mesh.vertices = vertices;
        mesh.triangles = triangles;
        mesh.normals = normals;
        mesh.uv = uv;
        mesh.RecalculateBounds ();
        //mesh.RecalculateNormals ();
        mesh_filter = (MeshFilter)terrainGo.AddComponent (typeof(MeshFilter));
        mesh_filter.mesh = mesh;
        mesh_collider = (MeshCollider)terrainGo.AddComponent (typeof(MeshCollider));
        mesh_collider.sharedMesh = mesh;
        mesh_renderer = (MeshRenderer)terrainGo.AddComponent (typeof(MeshRenderer));
        mesh_renderer.material = defaultMaterial;
        mesh_renderer.material.mainTexture = texture;
        texture.Apply ();
    }

    /**
   * *********************
   * Updates the mesh
   * *********************
   **/
    public void UpdateChunk ()
    {
        GenerateChunkData ();
        mesh.vertices = vertices;
        mesh.triangles = triangles;
        //mesh.RecalculateNormals ();
        mesh_collider.sharedMesh = null;
        mesh_collider.sharedMesh = mesh;
        mesh_filter.mesh = mesh;
```

```
        mesh.RecalculateBounds ();
        texture.Apply ();
    }

    /**
   * **********************
   * Moves the mesh to a new position
   * **********************
   **/
    public void setPosition (Vector3 newPosition)
    {
        pos = newPosition;
    }
}
```

A.6 MoveToCamera.cs

Listing A.6: MoveToCamera.cs

```
using UnityEngine;
using System.Collections;

public class MoveToCamera : MonoBehaviour {
  Camera cam;

  /**
   * **********************
   * Initialization
   * **********************
   **/
  void Start () {
    cam = Camera.main;
  }

  /**
   * **********************
   * Updates every frame
   * **********************
   **/
  void Update () {
    transform.position = cam.transform.position;
  }
}
```

A.7 PerlinNoise.cs

Listing A.7: PerlinNoise.cs

```
using UnityEngine;
using System.Collections;

public class PerlinNoise
{
    const int SIZE = 511;
    private int[] perm = new int[SIZE + SIZE];
    private static Vector2[] gradients2D = {
      new Vector2 (1f, 0f),
      new Vector2 (-1f, 0f),
      new Vector2 (0f, 1f),
      new Vector2 (0f, -1f),
      new Vector2 (1f, 1f).normalized,
      new Vector2 (-1f, 1f).normalized,
      new Vector2 (1f, -1f).normalized,
      new Vector2 (-1f, -1f).normalized
};
    private const int gradientsMask2D = 7;
    private static float sqr2 = Mathf.Sqrt (2f);

    /**
   * *********************
   * some code explanation
   * *********************
   **/
    public PerlinNoise (int seed)
    {
        UnityEngine.Random.seed = seed;

        int i, j, k;
        for (i = 0; i < SIZE; i++) {
            // creates 0 - 255
            perm [i] = i;
        }

        while (i > 1) {
            i--;
            k = perm [i];
            j = UnityEngine.Random.Range (0, SIZE);
            perm [i] = perm [j];
            perm [j] = k;
        }

        for (i = 0; i < SIZE; i++) {
            perm [SIZE + i] = perm [i];
        }
    }

    /**
   * *********************
   *  6t^5 - 15t^4 + 10t^3
   * *********************
   **/
```

```
    float Smooth (float t)
    {
        //return Mathf.Pow(t*t*t,2) - Mathf.Pow(t*t*t,3);;
        return t * t * t * (t * (t * 6.0f - 15.0f) + 10.0f);
    }

    /**
   * *********************
   * some code explanation
   * *********************
   **/
    private static float Dot (Vector2 g, float x, float y)
    {
        return g.x * x + g.y * y;
    }

    /**
   * *********************
   * some code explanation
   * *********************
   **/
    public float Perlin2D (Vector3 point, float frequency)
    {
        point *= frequency;
        int ix0 = Mathf.FloorToInt (point.x);
        int iy0 = Mathf.FloorToInt (point.y);
        float tx0 = point.x - ix0;
        float ty0 = point.y - iy0;
        float tx1 = tx0 - 1f;
        float ty1 = ty0 - 1f;
        ix0 &= SIZE;
        iy0 &= SIZE;
        int ix1 = (ix0 + 1) & SIZE;
        int iy1 = (iy0 + 1) & SIZE;

        Vector2 g00 = gradients2D [perm [perm [ix0] + iy0] &
            gradientsMask2D];
        Vector2 g10 = gradients2D [perm [perm [ix1] + iy0] &
            gradientsMask2D];
        Vector2 g01 = gradients2D [perm [perm [ix0] + iy1] &
            gradientsMask2D];
        Vector2 g11 = gradients2D [perm [perm [ix1] + iy1] &
            gradientsMask2D];

        float v00 = Dot (g00, tx0, ty0);
        float v10 = Dot (g10, tx1, ty0);
        float v01 = Dot (g01, tx0, ty1);
        float v11 = Dot (g11, tx1, ty1);

        float tx = Smooth (tx0);
        float ty = Smooth (ty0);
        return Mathf.Lerp (
          Mathf.Lerp (v00, v10, tx),
          Mathf.Lerp (v01, v11, tx),
          ty) * sqr2;
    }

```

```
    /**
   * *********************
   * Implementation of noise
   * *********************
   **/
    public float Noise (Vector2 point)
    {
        int x = Mathf.FloorToInt (point.x);
        int y = Mathf.FloorToInt (point.y);
        x &= SIZE;
        y &= SIZE;
        int v = perm [x + y];
        v &= SIZE;
        return v / 2;
    }


    /**
   * *********************
   * Implementation of 2D fractal perlin noise
   * *********************
   **/
    public float FractalNoise2D (Vector3 point, int octaves, float
        frequency, float lacunarity, float persistence, float gain)
    {
        float sum = Perlin2D (point, frequency);
        float amplitude = 1f;
        float range = 1f;
        for (int o = 1; o < octaves; o++) {
            frequency *= lacunarity;
            amplitude *= persistence;
            range += amplitude;
            sum += Perlin2D (point, frequency) * amplitude;
        }
        return (sum / range) * gain;
    }
}
```

A.8 PlayerCollision.cs

Listing A.8: PlayerCollision.cs

```
using UnityEngine;
using System.Collections;

public class PlayerCollision : MonoBehaviour
{
    public GameManager gm;
    private PlayerSound ps;
    Vector3 chunkPosition = new Vector3 (0, 0, 0);

    /**
   * *********************
   * Initialization
   * *********************
   **/
    void Start ()
    {
        ps = gameObject.GetComponent<PlayerSound> ();
    }

    /**
   * *********************
   * Used for collecting coins
   * *********************
   **/
    void OnControllerColliderHit (ControllerColliderHit hit)
    {
        if (hit.gameObject.name == "terrainChunk") {
            chunkPosition = hit.gameObject.transform.position;
        }

        if (hit.gameObject.name == "Coin(Clone)") {
            gm.AddPoints (1);
            gm.CollectCoin (chunkPosition);
            ps.PlayCoinSound ();
            Destroy (hit.gameObject);
        }
    }
}
```

A.9 PlayerSound.cs

Listing A.9: PlayerSound.cs

```
using UnityEngine;
using System.Collections;

public class PlayerSound : MonoBehaviour
{
    public AudioClip[] footstepSounds;
    public AudioClip[] jumpSounds;
    public AudioClip gunSound;
    public AudioClip coinSound;
    public int points = 0;
    private CharacterController pc;
    private bool isWalking = false;
    private bool isFalling = false;
    public float walkSpeed = 0.3f;

    /**
    * **********************
    * Initialization
    * **********************
    **/
    void Start ()
    {
        pc = GetComponent<CharacterController> ();
        InvokeRepeating ("WalkSound", 0.0f, walkSpeed);
    }

    /**
    * **********************
    * Updates every frame
    * **********************
    **/
    void Update ()
    {
        if (!pc.isGrounded) {
            isFalling = true;
        }
        if (pc.velocity.magnitude > 0.3f && pc.isGrounded) {
            isWalking = true;
        } else {
            isWalking = false;
        }

        if (Input.GetKeyDown ("space") && pc.isGrounded) {
            PlayJumpSound ();
        }
    }

    /**
    * **********************
    * Play walk sound
    * **********************
    **/
    void WalkSound ()
```

```
    {
        if (isWalking) {
            audio.pitch = Random.Range (0.9f, 1.1f);
            audio.PlayOneShot (footstepSounds [(int)Random.Range
                (0, footstepSounds.Length)], 0.5f);

        }
    }

    /**
   * *********************
   * Play jump sound
   * *********************
   **/
    void PlayJumpSound ()
    {
        audio.pitch = Random.Range (0.9f, 1.1f);
        audio.PlayOneShot (jumpSounds [(int)Random.Range (0,
            jumpSounds.Length)], 1f);
    }

    /**
   * *********************
   * Play collect coin sound
   * *********************
   **/
    public void PlayCoinSound ()
    {
        audio.pitch = Random.Range (0.7f, 1.3f);
        audio.PlayOneShot (coinSound, 1f);
    }
}
```

A.10 Rotation.cs

Listing A.10: Rotation.cs

```
using UnityEngine;
using System.Collections;

public class Rotation : MonoBehaviour
{
    public float speed = 0.4f;
    public float startangle = 272;
    private float yRotation;
    private float xRotation;

    /**
   * *********************
   * Initialization
   * *********************
   **/
    void Start ()
    {
        yRotation = startangle;
    }

    /**
   * *********************
   * Updates every frame
   * *********************
   **/
    void Update ()
    {
        transform.rotation = Quaternion.Euler (new Vector3 (
            yRotation, 0, xRotation));
        yRotation = yRotation + speed;
        xRotation = xRotation + speed;
    }
}
```

A.11 Tree2.cs

Listing A.11: Rotation.cs

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

/***************
* This code is used to generate a tree based on L-System Algorithm
* It was initialy developed by Chanfort, a unity forum user (http
    ://forum.unity3d.com/members/chanfort.503785/)
* Source : http://forum.unity3d.com/threads/l-systems-for-unity-
    free-script-included.272416/
* We modified this code in order to better fitting with our
    expectation for our project.
****************/

public class Tree2 : MonoBehaviour
{

    //List for the number of segments
    private List<int> numberSegments = new List<int> ();
    public float segBottomRadius = .55f;
    public float segTopRadius = .15f;
    public float segLength = 1.0f;

    //All the list use for vertices, normales, uv, triangle, etc
        ...
    private List<float> curBotRadius = new List<float> ();
    private List<float> curTopRadius = new List<float> ();
    private List<Vector3> gvertices = new List<Vector3> ();
    private List<Vector3> gnormals = new List<Vector3> ();
    private List<Vector2> guvs = new List<Vector2> ();
    private List<int> gtriangles = new List<int> ();
    private List<int> topVertices = new List<int> ();
    private List<int> minVertex = new List<int> ();
    private List<int> maxVertex = new List<int> ();
    private List<int> minTriangle = new List<int> ();
    private List<int> maxTriangle = new List<int> ();

    //All the others initials parameters
    private int currentSegmentId = 0;
    private int currentSegmentOffset = 0;
    private int currentBranchId = 0;
    private int verticesOffset = 0;
    private int trianglesOffset = 0;
    private List<float> angle = new List<float> ();
    public float iniAngle = 25.0f;
    private List<Vector3> segmentPos = new List<Vector3> ();
    private List<Quaternion> segmentRot = new List<Quaternion> ();
    private List<Vector3> segmentRotV = new List<Vector3> ();
    private List<Vector3> segmentLocRotVect = new List<Vector3> ();
    private int nBranchesToAdd = 0;
    private int nbSides = 18;
    private List<int> inhSegId = new List<int> ();
    private List<int> inhBranchId = new List<int> ();
```

```
    private List<int> branchingOrder = new List<int> ();
    private List<Vector3> iniPos3 = new List<Vector3> ();
    private Vector3 iniPos;
    private Vector3 iniPos2;

    //The gameobject, the mesh, filter and renderer
    public GameObject plane;
    public MeshFilter filter;
    public Mesh mesh;
    public MeshRenderer renderer;

    /***************
   * Controllabilty
  **************/

    public int numberSegmentsOrigin = 15; //Number of maximum segments = number of iterations
    public float coeffAngleBranch = 4.0f; //Coeff Angle Branch
    public float coeffBranchPossibility = 1.0f; // Coeff Branch Possibility
    public int numberSegmentTrunk = 5; // Number of segment for the trunk
    public int numberSegmentFirstBranch = 2; // Number of segment before the first branch

    /***************
   * Setup the Cone for the tree
  **************/
    public void SetupCone (int seed, int numberSegmentsOrigin, float coeffAngleBranch, float coeffBranchPossibility, int numberSegmentTrunk, int numberSegmentFirstBranch)
    {

        this.numberSegmentsOrigin = numberSegmentsOrigin;
        this.coeffAngleBranch = coeffAngleBranch;
        this.coeffBranchPossibility = coeffBranchPossibility;
        this.numberSegmentTrunk = numberSegmentTrunk;
        this.numberSegmentFirstBranch = numberSegmentFirstBranch;

        //Assign the seed for controllability of the random parts
        Random.seed = seed;
        //Add the number maximum of segments
        numberSegments.Add (numberSegmentsOrigin);

        //Initials parameters for the tree
        segBottomRadius = Random.Range (0.35f, 0.55f);
        segTopRadius = Random.Range (0.12f, 0.17f);
        segLength = Random.Range (0.5f, 0.7f);
        iniAngle = 0.0f / segLength;
        angle.Add (0.0f);
        curBotRadius.Add (0f);
        curTopRadius.Add (0f);
        inhSegId.Add (currentSegmentId);
        inhBranchId.Add (0);
        branchingOrder.Add (0);
        iniPos = new Vector3 (0f, segLength, 0f);
        iniPos2 = new Vector3 (0f, segLength, 0f);
```

```
        iniPos3.Add (iniPos);
        Vector3 rotVect = new Vector3 (0f, 0f, 1f);
        segmentPos.Add (new Vector3 (0f, 0f, 0f));
        segmentLocRotVect.Add (new Vector3 (0f, 1f, 0f));
        segmentRot.Add (Quaternion.AngleAxis (iniAngle, rotVect));
        segmentRotV.Add (new Vector3 (0f, 0f, 0f));
        curBotRadius [currentBranchId] = segBottomRadius;
        curTopRadius [currentBranchId] = segBottomRadius - (
            segBottomRadius - segTopRadius) / numberSegments [
            currentBranchId];

        //Drawing the first cone
        DrawCone ();
        //Assign branch possibility
        float branchPossibility = coeffBranchPossibility;
        //Calcul the rotation vector
        rotVect = Vector3.Cross
      (
        (segmentLocRotVect [currentBranchId]),
        new Vector3 (Random.Range (-1, 1f), Random.Range (-1, 1f),
            Random.Range (-1, 1f))
        ).normalized;


        //CONSTRUCT THE BASE OF THE TREE
        angle [currentBranchId] = angle [currentBranchId] +
            iniAngle;
        segmentRot [currentBranchId] = Quaternion.AngleAxis (angle
            [currentBranchId], rotVect);
        segmentRotV [currentBranchId] = RotVector (iniAngle,
            segmentRotV [currentBranchId], rotVect);
        segmentPos [currentBranchId] = segmentPos [currentBranchId]
             + (segmentRot [currentBranchId] * iniPos);
        currentSegmentOffset = 0;
        curTopRadius [currentBranchId] = segBottomRadius - (
            segBottomRadius - segTopRadius) * (2) / numberSegments
            [currentBranchId];
        //Draw a cone for the base of the tree
        DrawCone ();

        for (int i=minVertex[currentSegmentId-1]; i<maxVertex[
            currentSegmentId-1]; i++) {
            if (topVertices [i] == 1) {
                gvertices [i] = gvertices [i - currentSegmentOffset
                     - 1];
            }
        }
        //Add a branch
        AddBranch ();
        //One less segment to construct
        numberSegments.Add (numberSegments [currentBranchId] - 1);
        int branchIsLocked = 0;


        //While there is a branch to add
        while (nBranchesToAdd>0) {
            currentBranchId++;
```

```
            //Calcul the angle for this branch
            Vector3 randVect = new Vector3 (Random.Range (-1, 1f),
                Random.Range (-1, 1f), Random.Range (-1, 1f));
            rotVect = Vector3.Cross (segmentLocRotVect [
                currentBranchId], randVect).normalized;
            angle [currentBranchId] = 10f;

            //COEFFICIENT TO INFLUENCE THE ANGLE OF THE BRANCH
            iniAngle = coeffAngleBranch / segLength * ((iniAngle +
                0.001f) / Mathf.Abs ((iniAngle + 0.001f)));
            if (inhBranchId [currentBranchId] > 0) {
                segmentRot [currentBranchId] = (Quaternion.
                    AngleAxis (0.5f * iniAngle, rotVect));
                segmentRotV [currentBranchId] = RotVector (iniAngle
                    , segmentRotV [currentBranchId], rotVect);
            } else {

            }
            segBottomRadius = curTopRadius [currentBranchId];
            //Draw a cone
            DrawCone ();
            for (int i=minVertex[currentSegmentId-1]; i<maxVertex[
                currentSegmentId-1]; i++) {
                if (topVertices [i] == 1) {
                    gvertices [i] = gvertices [i - (minVertex [
                        currentSegmentId - 1] - minVertex [inhSegId
                         [currentBranchId]])];
                }
            }
            iniPos2 = iniPos3 [currentBranchId];
            //Assign branch possibility
            branchPossibility = coeffBranchPossibility;

            for (int j=1; j<numberSegments[currentBranchId]; j++) {
                if (branchIsLocked == 0) {
                    angle [currentBranchId] = angle [
                        currentBranchId] + iniAngle;
                    segmentRot [currentBranchId] = segmentRot [
                        currentBranchId] * Quaternion.AngleAxis (
                        angle [currentBranchId], rotVect);
                    segmentRotV [currentBranchId] = RotVector (
                        iniAngle, segmentRotV [currentBranchId],
                        rotVect);

                    //NUMBER OF SEGEMENT FOR THE TRUNK
                    if (j < numberSegmentTrunk) {
                        iniPos2 = RotVector (0, iniPos2, rotVect);
                    } else {
                        iniPos2 = RotVector (iniAngle, iniPos2,
                            rotVect);
                    }

                    segmentPos [currentBranchId] = segmentPos [
                        currentBranchId] + RotVector (iniAngle,
                        iniPos2, rotVect);
                    currentSegmentOffset = 0;
```

```
                  curTopRadius [currentBranchId] = segBottomRadius - (segBottomRadius - segTopRadius) * (j + 1) / numberSegments [currentBranchId];

                  //it will get more chance to create a branch
                  branchPossibility = branchPossibility - 0.1f;
                  //Do we need to create branches
                  if ((j > numberSegmentFirstBranch) && (Random.Range (0f, 1f) > branchPossibility)) {

                      //Add a branch
                      AddBranch ();
                      numberSegments.Add (numberSegments [currentBranchId] - j);
                      //Assign branch possibility
                      branchPossibility = coeffBranchPossibility;

                      //Add a branch
                      AddBranch ();
                      numberSegments.Add (numberSegments [currentBranchId] - j);
                      //Assign branch possibility
                      branchPossibility = coeffBranchPossibility;

                      branchIsLocked = 0;
                  }

                  DrawCone ();

                  for (int i=minVertex[currentSegmentId-1]; i< maxVertex[currentSegmentId-1]; i++) {
                      if (topVertices [i] == 1) {
                          gvertices [i] = gvertices [i - currentSegmentOffset - 1];
                      }
                  }
              }
          }
          branchIsLocked = 0;
          nBranchesToAdd--;
      }

  }

  /***************
 * Get Normals
**************/
  Vector3 GetNormal (Vector3 a, Vector3 b, Vector3 c)
  {
      Vector3 side1 = b - a;
      Vector3 side2 = c - a;
      return Vector3.Cross (side1, side2).normalized;
  }

  /***************
 * Rotate Vector
```

```
**************/
  Vector3 RotVector (float rotAngle, Vector3 original, Vector3
      direction)
  {
      Vector3 cross1 = Vector3.Cross (original, direction);
      Vector3 cross2 = Vector3.Cross (original, cross1);
      Vector3 rotatedVector = Quaternion.AngleAxis (rotAngle,
          cross2) * original;
      return rotatedVector;
  }

  /***************
 * Add  a new Branch
**************/
  void AddBranch ()
  {
      inhSegId.Add (currentSegmentId);
      inhBranchId.Add (currentBranchId);
      curBotRadius.Add (curBotRadius [currentBranchId]);
      curTopRadius.Add (curTopRadius [currentBranchId]);
      segmentPos.Add (segmentPos [currentBranchId]);
      segmentRot.Add (segmentRot [currentBranchId]);
      segmentRotV.Add (segmentRotV [currentBranchId]);
      segmentLocRotVect.Add (segmentLocRotVect [currentBranchId])
          ;
      angle.Add (0f);
      nBranchesToAdd++;
      branchingOrder.Add (branchingOrder [currentBranchId] + 1);
      iniPos3.Add (iniPos2);
  }


  /***************
 * Draw Cone
**************/
  void DrawCone ()
  {
      int useBottomCap = 0;
      int useTopCap = 0;
      float height = segLength;
      float bottomRadius = curBotRadius [currentBranchId];
      float topRadius = curTopRadius [currentBranchId];
      int nbVerticesCap = nbSides + 1;

      /***************
   * Construction of the vertices
  **************/
      int NN = nbVerticesCap + nbVerticesCap + nbSides * 2 + 2;
      int[] vUsed = new int[NN];
      int[] isTopVertice = new int[NN];
      int numUsed = 0;

      for (int ii=0; ii<NN; ii++) {
          vUsed [ii] = 0;
          isTopVertice [ii] = 0;
      }

```

```

        // total number of vertices needed
        Vector3[] vertices = new Vector3[nbVerticesCap +
            nbVerticesCap + nbSides * 2 + 2];
        int vert = 0;
        float _2pi = Mathf.PI * 2f;

        // construction of the bottom vertices
        if (useBottomCap == 1) {
            vert = 0;
            vUsed [vert] = 1;
            vertices [vert++] = new Vector3 (0f, 0f, 0f);
            numUsed++;
            while (vert <= nbSides) {
                float rad = (float)vert / nbSides * _2pi;
                vertices [vert] = new Vector3 (Mathf.Cos (rad) *
                    bottomRadius, 0f, Mathf.Sin (rad) *
                    bottomRadius);
                vUsed [vert] = 1;

                numUsed++;
                vert++;
            }
        }

        // construction of the top vertices
        if (useTopCap == 1) {
            vert = nbSides + 1;
            vUsed [vert] = 1;
            vertices [vert++] = new Vector3 (0f, height, 0f);
            numUsed++;
            while (vert <= nbSides * 2 + 1) {
                float rad = (float)(vert - nbSides - 1) / nbSides *
                     _2pi;
                vertices [vert] = new Vector3 (Mathf.Cos (rad) *
                    topRadius, height, Mathf.Sin (rad) * topRadius)
                    ;
                vUsed [vert] = 1;
                numUsed++;
                vert++;
            }
        }

        vert = nbSides * 2 + 2;
        int v = 0;

        // construction of the side vertices
        while (vert <= vertices.Length - 4) {
            float rad = (float)v / nbSides * _2pi;
            vertices [vert] = new Vector3 (Mathf.Cos (rad) *
                topRadius, height, Mathf.Sin (rad) * topRadius);
            vertices [vert + 1] = new Vector3 (Mathf.Cos (rad) *
                bottomRadius, 0, Mathf.Sin (rad) * bottomRadius);

            isTopVertice [vert + 1] = 1;

            vUsed [vert] = 1;
```

```
            vUsed [vert + 1] = 1;

            vert += 2;
            numUsed += 2;
            v++;
        }
        vertices [vert] = vertices [nbSides * 2 + 2];
        vertices [vert + 1] = vertices [nbSides * 2 + 3];

        isTopVertice [vert + 1] = 1;

        vUsed [vert] = 1;
        vUsed [vert + 1] = 1;

        Vector3[] verticesA = new Vector3[numUsed + 2];

        int jj = 0;
        for (int ii=0; ii<NN; ii++) {
            if (vUsed [ii] == 1) {
                verticesA [jj] = vertices [ii];
                vertices [ii] = segmentRotV [currentBranchId] +
                    vertices [ii];
                gvertices.Add (vertices [ii] + segmentPos [
                    currentBranchId]);
                jj++;
            }
        }
        segmentLocRotVect [currentBranchId] = (segmentRotV [
            currentBranchId] + (new Vector3 (0f, 1f, 0f) - new
            Vector3 (0f, 0f, 0f))).normalized;
        jj = 0;
        for (int ii=0; ii<NN; ii++) {
            if (vUsed [ii] == 1) {

                if (isTopVertice [ii] == 1) {
                    topVertices.Add (1);
                } else {
                    topVertices.Add (0);
                }

                jj++;
            }
        }

        /***************
     * Construction of the normals
    **************/

        Vector3[] normalsA = new Vector3[verticesA.Length];
        Vector3[] normals = new Vector3[vertices.Length];
        numUsed = 0;
        vert = 0;
        for (int ii=0; ii<NN; ii++) {
            vUsed [ii] = 0;
        }

        // Construction of the bottom normals (down)
```

```
        if (useBottomCap == 1) {
            vert = 0;
            while (vert  <= nbSides) {
                normals [vert] = Vector3.down;
                vUsed [vert] = 1;
                numUsed++;
                vert++;
            }
        }

        // Construction of the top normals (up)
        if (useTopCap == 1) {
            vert = nbSides + 1;
            while (vert <= nbSides * 2 + 1) {
                normals [vert] = Vector3.up;
                vUsed [vert] = 1;
                numUsed++;
                vert++;
            }
        }

        vert = nbSides * 2 + 2;
        v = 0;

        // Construction of the side normals
        while (vert <= vertices.Length - 4) {
            float rad = (float)v / nbSides * _2pi;
            float cos = Mathf.Cos (rad);
            float sin = Mathf.Sin (rad);
            normals [vert] = new Vector3 (cos, 0f, sin);
            normals [vert + 1] = normals [vert];
            vUsed [vert] = 1;
            vUsed [vert + 1] = 1;
            numUsed += 2;
            vert += 2;
            v++;
        }
        normals [vert] = normals [nbSides * 2 + 2];
        normals [vert + 1] = normals [nbSides * 2 + 3];
        vUsed [vert] = 1;
        vUsed [vert + 1] = 1;

        jj = 0;
        for (int ii=0; ii<NN; ii++) {
            if (vUsed [ii] == 1) {
                normalsA [jj] = normals [ii];
                gnormals.Add (normals [ii]);
                jj++;
            }
        }

        /***************
      * Construction of the UV
     **************/
        Vector2[] uvsA = new Vector2[verticesA.Length];
        Vector2[] uvs = new Vector2[vertices.Length];
        int[] uUsed = new int[vertices.Length];
```

```
        for (int ii=0; ii<NN; ii++) {
            uUsed [ii] = 0;
        }

        numUsed = 0;
        int u = 0;

        // Construction of the bottom uv
        if (useBottomCap == 1) {
            u = 0;

            uUsed [u] = 1;
            uvs [u++] = new Vector2 (0.5f, 0.5f);
            numUsed++;

            while (u <= nbSides) {
                float rad = (float)u / nbSides * _2pi;
                uvs [u] = new Vector2 (Mathf.Cos (rad) * .5f + .5f,
                     Mathf.Sin (rad) * .5f + .5f);
                uUsed [u] = 1;
                numUsed++;
                u++;
            }
        }

        // Construction of the top uv
        if (useTopCap == 1) {
            u = nbSides + 1;

            uUsed [u] = 1;
            uvs [u++] = new Vector2 (0.5f, 0.5f);
            numUsed++;
            while (u <= nbSides * 2 + 1) {
                float rad = (float)u / nbSides * _2pi;
                uvs [u] = new Vector2 (Mathf.Cos (rad) * .5f + .5f,
                     Mathf.Sin (rad) * .5f + .5f);
                uUsed [u] = 1;
                numUsed++;
                u++;
            }
        }

        u = nbSides * 2 + 2;
        int u_sides = 0;

        // Construction of the sides uv
        while (u <= uvs.Length - 4) {
            float t = (float)u_sides / nbSides;
            uvs [u] = new Vector3 (t, 1f);
            uvs [u + 1] = new Vector3 (t, 0f);
            uUsed [u] = 1;
            uUsed [u + 1] = 1;
            numUsed += 2;
            u += 2;
            u_sides++;
        }
        uvs [u] = new Vector2 (1f, 1f);
```

```
        uvs [u + 1] = new Vector2 (1f, 0f);
        uUsed [u] = 1;
        uUsed [u + 1] = 1;

        jj = 0;
        for (int ii=0; ii<NN; ii++) {
            if (uUsed [ii] == 1) {
                uvsA [jj] = uvs [ii];
                guvs.Add (uvs [ii]);
                jj++;
            }
        }

        /***************
     * Construction of the triangles
    **************/

        int nbTriangles = nbSides + nbSides + nbSides * 2;
        int[] triangles = new int[nbTriangles * 3 + 3];
        int NT = nbTriangles * 3 + 3;
        int[] tUsed = new int[NT];

        for (int ii=0; ii<NT; ii++) {
            tUsed [ii] = 0;
        }

        numUsed = 0;

        int tri = 0;
        int i = 0;
        int missTris = 0;
        // Construction of the bottom triangles
        if (useBottomCap == 1) {
            while (tri < nbSides - 1) {
                if (useBottomCap == 1) {
                    triangles [i] = 0;
                    triangles [i + 1] = tri - missTris + 1;
                    triangles [i + 2] = tri - missTris + 2;

                    tUsed [i] = 1;
                    tUsed [i + 1] = 1;
                    tUsed [i + 2] = 1;
                }
                numUsed += 3;
                tri++;
                i += 3;
            }
            triangles [i] = 0;
            triangles [i + 1] = tri - missTris + 1;
            triangles [i + 2] = 1;

            tUsed [i] = 1;
            tUsed [i + 1] = 1;
            tUsed [i + 2] = 1;
            numUsed += 3;
            tri++;
            i += 3;
```

```
        }

        if (useBottomCap == 1) {
            if (useTopCap == 1) {
                missTris = 0;
            }
            if (useTopCap == 0) {
                missTris = 0;
            }
        } else if (useBottomCap == 0) {
            if (useTopCap == 1) {
                missTris = nbSides + 1;
            } else if (useTopCap == 0) {
                missTris = nbSides + 1;
            }
        }

        tri = nbSides;
        i = 3 * tri;

        // Construction of the top triangles
        if (useTopCap == 1) {
            while (tri < nbSides*2) {

                triangles [i] = tri - missTris + 2;
                triangles [i + 1] = tri - missTris + 1;
                triangles [i + 2] = nbVerticesCap - missTris;

                tUsed [i] = 1;
                tUsed [i + 1] = 1;
                tUsed [i + 2] = 1;

                numUsed += 3;

                tri++;
                i += 3;

            }

            triangles [i] = nbVerticesCap - missTris + 1;
            triangles [i + 1] = tri - missTris + 1;
            triangles [i + 2] = nbVerticesCap - missTris;

            tUsed [i] = 1;
            tUsed [i + 1] = 1;
            tUsed [i + 2] = 1;

            numUsed += 3;
            tri++;
            i += 3;
            tri++;
        }

        if (useBottomCap == 1) {
            if (useTopCap == 1) {
                missTris = 0;
            }
```

```
            if (useTopCap == 0) {
                missTris = nbSides + 1;
            }
        } else if (useBottomCap == 0) {
            if (useTopCap == 1) {
                missTris = nbSides + 1;
            } else if (useTopCap == 0) {
                missTris = nbSides * 2 + 2;
            }
        }

        tri = nbSides * 2 + 2;
        i = 3 * tri - 3;


        // Construction of the sides triangles
        while (tri <= nbTriangles) {
            triangles [i] = tri - missTris + 2;
            triangles [i + 1] = tri - missTris + 1;
            triangles [i + 2] = tri - missTris + 0;

            tUsed [i] = 1;
            tUsed [i + 1] = 1;
            tUsed [i + 2] = 1;

            numUsed += 3;
            tri++;
            i += 3;

            triangles [i] = tri - missTris + 1;
            triangles [i + 1] = tri - missTris + 2;
            triangles [i + 2] = tri - missTris + 0;

            tUsed [i] = 1;
            tUsed [i + 1] = 1;
            tUsed [i + 2] = 1;

            numUsed += 3;
            tri++;
            i += 3;
        }

        int[] trianglesA = new int[numUsed];

        jj = 0;
        for (int ii=0; ii<NT; ii++) {
            if (tUsed [ii] == 1) {
                trianglesA [jj] = triangles [ii];
                gtriangles.Add (triangles [ii] + verticesOffset);
                jj++;
            }
        }

        jj = 0;
        minVertex.Add (verticesOffset);
        for (int ii=0; ii<NN; ii++) {

```

```
            if (vUsed [ii] == 1) {

                jj++;
            }
        }
        maxVertex.Add (verticesOffset + jj);
        verticesOffset = verticesOffset + jj;
        currentSegmentOffset = jj;

        minTriangle.Add (trianglesOffset);
        jj = 0;
        for (int ii=0; ii<NT; ii++) {
            if (tUsed [ii] == 1) {

                jj++;
            }
        }
        maxTriangle.Add (trianglesOffset + jj);
        trianglesOffset = trianglesOffset + jj;
        currentSegmentId++;
    }


    /***************
     * Construction of the mesh
    **************/
    public void CreateMesh ()
    {
        //Creating the gameobject
        plane = new GameObject ("Tree2");
        //Adding a mesh filter component
        filter = plane.AddComponent<MeshFilter> ();
        mesh = filter.mesh;
        //Clear the mesh
        mesh.Clear ();
        //Adding vertices, normales, uv, triangless
        mesh.vertices = gvertices.ToArray ();
        mesh.normals = gnormals.ToArray ();
        mesh.uv = guvs.ToArray ();
        mesh.triangles = gtriangles.ToArray ();
        mesh.RecalculateBounds ();
        //Render it
        renderer = plane.AddComponent (typeof(MeshRenderer)) as
            MeshRenderer;
        renderer.material.shader = Shader.Find ("Toon/Lighted
            Outline");
        Texture2D tex = new Texture2D (1, 1);
        tex.SetPixel (0, 0, Color.grey);
        tex.Apply ();
        renderer.material.color = Color.grey;
    }

    /***************
     * Destruction of the mesh
    **************/
    public void DestroySelf ()
    {
```

```
        Destroy (plane);
        Destroy (gameObject);
    }
}
```

A.12 FireFly.cs

Listing A.12: Rotation.cs

```
using UnityEngine;
using System.Collections;

public class FireFly : MonoBehaviour
{
    float x;
    float y;
    float z;
    float x_speed = 0;
    float y_speed = 0;
    float z_speed = 0;
    private int[] scale;
    public AudioClip fireflyBell;
    public GameObject target;

    /**
   * *********************
   * Initialization
   * *********************
   **/
    void Start ()
    {
        target = GameObject.Find ("Graphics");
        x = transform.position.x;
        y = transform.position.y;
        z = transform.position.z;
        InvokeRepeating ("changeDirection", 0, 0.2F);
        InvokeRepeating ("PlayBell", 0.0f, Random.Range (3f, 5f));
    }

    /**
   * *********************
   * Updates every frame
   * *********************
   **/
    void Update ()
    {
        x = x + x_speed;
        y = y + y_speed;
        z = z + y_speed;
        transform.position = new Vector3 (x, y, z);
        if (y < 1) {
            y = 1;
        }
        if (y > 4) {
            y = 4;
        }
        target = GameObject.Find ("Graphics");
        transform.rotation = transform.localRotation;
    }
    /**
   * *********************
   * Play bell sounds
```

```
    * *********************
    **/
     void PlayBell ()
     {
         int[] scale = new int[8] {0,2,4,6,7,9,11,12};
         float pitchIndex = scale [Random.Range (0, 7)];
         float pitch = pitchCorrect (pitchIndex);
         audio.pitch = pitch;
         audio.PlayOneShot (fireflyBell, 1f);
     }
     /**
    * *********************
    * Change Direction randomly
    * *********************
    **/
     void changeDirection ()
     {
         x_speed = Random.Range (-0, 0.02f) - 0.01f;
         y_speed = Random.Range (-0f, 0.02f) - 0.01f;
         z_speed = Random.Range (-01f, 0.02f) - 0.01f;
     }

     /**
    * *********************
    * DestroySelf
    * *********************
    **/
     public void DestroySelf ()
     {
         Destroy (gameObject);
     }

     /**
    * *********************
    * Speed -> pitch
    * *********************
    **/
     public float pitchCorrect (float speed)
     {
         return Mathf.Pow (2, speed / 12.0f);
     }
}
```

A.13 Coin.cs

Listing A.13: Rotation.cs

```
using UnityEngine;
using System.Collections;
/**
   * *********************
   * Used for coins, which have no need for implementation at the
       moment
   * *********************
   **/
public class Coin : GenericObject {


}
```

Appendix B

Instruction guide

All the attached programs can be executed on Windows, Mac OS X and Linux based systems. In order to run the programs, simply use the executable file.

B.1 Surogou

When the program is executed, the programs enters the title screen. In the title screen the user can adjust the draw distance in the world, by using the *"draw distance"* slider. In the *"seed"* text box the user can enter a string of letters, that will be used for the seed. Use the button *"Generate World"* in order to start the game. In order to exit the game or generate a new world, use the *"escape"* key.

Movement

To explore the world use the *"w"* and *"s"* key on the keyboard to move forward and backward and the *"a"* and *"d"* key to strafe left and right. The mouse or keypad is used to look around. The *"space"* key can be used for jumping, if the user needs to elevate to higher terrain.

B.2 Surogou (Unity Project)

The *Unity* project for Surogou is attached to the project. In order to run this project, *Unity Free*, which can be downloaded on *"http://unity3d.com/"*, has to be installed on the system. The main scene for the project can be found in the folder: "2 Unity Project/16122014/Assets/InfiniteProblem02.unity".

B.3 Perlin Tester

This program is a special version of *Surogou*, where the user can create a terrain using the the six parameters: octave, frequency, lacunarity, persistence, gain and type, described on page 76.

B.4 L-System Tester

L-System Tester, is used to test our modified version of *Chanforts* [3] L-Tree algorithm. The parameters that can be used are descibed on page 62.

B.5 Infinite World

Infinite World is the program which is described on page 25. This program shows, how position of the player can be used to create a consistent game world. A technique we also apply in *Surogou*. The arrow keys of the keyboard are used to move the view of the player. Note: there are no way to exit this program, therefore it is recommended to run the program in window mode.