Arttu Marttinen

Procedural Generation of Two-Dimensional Levels

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

15 October 2017

Author Title Number of Pages Date	Arttu Marttinen Procedural Generation of Two-Dimensional Levels 28 pages 15 October 2017
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Game Applications
Instructors	Miikka Mäki-Uuro, Senior Lecturer

This Bachelor’s Thesis investigates level generation methods of two-dimensional games. The goal of this thesis is to create a two-dimensional level generation algorithm for an action role-playing game. The game itself will be progress-oriented but exploration should also be a great part of the gameplay.

The game’s specifications come with certain requirements for the two-dimensional generation. The algorithm should be able to create a level with cave networks of modifiable sizes. In addition, the algorithm must be able to generate biomes that are not directly related to their position in the world.

It is desirable to have the level generation be as optimized as possible, however, there is no set time limit for the generation. This decision was made because of the sheer size of the generated levels is relatively large and the levels are not being generated often in a game such as this.

The conclusion regarding which algorithms to use was made after taking multiple ones into consideration and testing their flexibility. The terrain generation was implemented using as few algorithms as possible.

The algorithm created in this thesis fulfills the requirements. It yields levels that have vast differences, yet share some fundamental characteristics. The cave and biome generations are separate, meaning that the generation of one does not affect the other. The levels have clear cave networks and they look generally pleasant.

Keywords	Perlin noise, game development, level generation

Tekijä Otsikko Sivumäärä Aika	Arttu Marttinen 2D-tasojen proseduraalinen generointi 28 sivua 15.10.2017
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Pelisovellukset
Ohjaajat	Lehtori, Miikka Mäki-Uuro

Tässä työssä tutkitaan 2D-pelien tasogenerointiin käytettäviä metodeja. Työn tavoitteena on oman tasogenerointi-algoritmin kehittäminen 2D-pelille.

Pelin omaamat ominaisuudet asettavat tiettyjä vaatimuksia tälle algoritmille. Algoritmin tulisi kyetä luomaan tasoja joista löytyy merkittäviä luolaverkostoja. Luolaverkostojen tulisi myös omata useampi muokattavissa oleva tekijä, kuten luolien pituus. Algoritmin tulisi myös tukea biomien luontia joiden tyyppi ei ole suoraan yhteydessä niiden sijaintiin maailmassa.

Algoritmin olisi suotavaa olla mahdollisimman optimoitu, tämä ei kuitenkaan ole sen elinehto johtuen pelin tyypistä. Tähän päätökseen päädyttiin johtuen tasojen laajuudesta ja siitä ettei tämänkaltaisessa pelissä generoida tasoja usein.

Algoritmin hyödyntämät ns. ”alialgoritmit” valittiin useamman algoritmin evaluoinnin jälkeen. Alialgoritmien määrä pyrittiin pitämään mahdollisimman vähäisenä ja siinä onnistuttiin kiitettävän hyvin.

Kehitetty algoritmi täyttää sille asetetut vaatimukset luoden näyttäviä luolaverkostoja jotka omaavat useampia muokattavia arvoja. Algoritmin biomien generointi ei ole sidottu biomien sijaintiin millään lailla ja näiden generointi omaa myös omat muokattavat arvonsa. Voidaan siis todeta algoritmin olevan alkuperäiset vaatimukset täyttävä.

Avainsanat	Perlin-kohina, pelikehitys, tasogenerointi

Contents

1 Introduction

This Bachelor’s Thesis explores the implementation of a procedural two-dimensional level generation algorithm. Procedural generation itself refers to a generation style that uses algorithms to generate the values as opposed to manually coming up with them.

The algorithm itself must meet certain requirements, including that each iteration must have noticeable differences, yet share some characteristics. There must be common characteristics between iterations for the game to maintain its look and feel.

The game itself will be endlessly expandable and because of this the algorithm should also work in a way that it will allow additional content generation to be made later. It is desirable that a game of this nature has procedurally generated levels for the player to have an immense number of worlds to explore.

Chapter two of this work dive into describes the details of the desired outcome, i.e. the desired level type. Chapter three focuses on the evaluation of the sub-algorithms. Chapter four delves into the technical details of the implemented algorithm. Chapter five explains the features of the algorithm. Chapter six takes a critical look at the faults of the developed algorithm. Chapter seven explains other use cases of procedural generation. Chapter eight provides a summary of the Bachelor’s Thesis.

2 Goals of the algorithm

When the goals were initially set technical limitations were not considered as it was clear that they would have to be factored in later. Taking technical limitations into account from the very first step of the design would end up narrowing the design much more than ignoring it at first would.

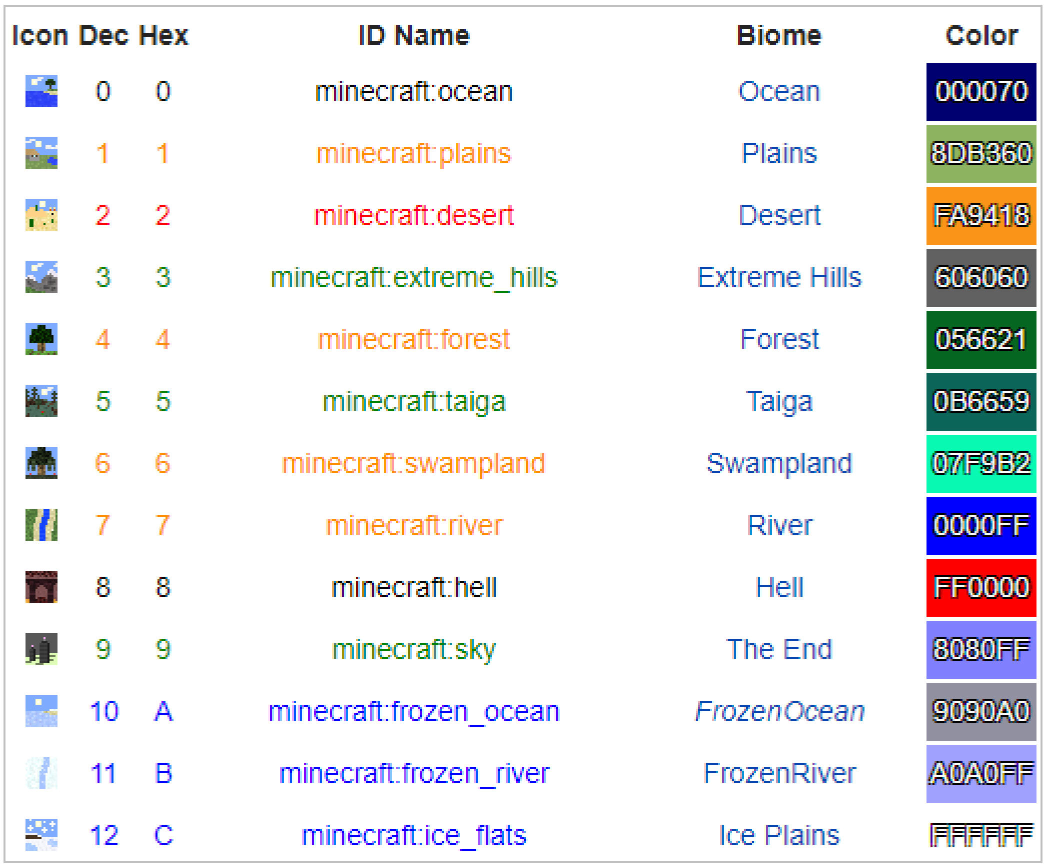
This chapter goes over the initial decisions that were made and iterated upon.

2.1 Reality vs Playability

Creating realistic terrain is often desirable, however this is not always the case. For example, in this game it is desirable to have flat plateaus surrounding the player's initial spawn location. Having the spawn area be in a relatively flat area makes it easy and safe for the player to get used to the controls and have room to explore.

This requires that the algorithm can generate terrain that is at least partially flat. However, it would be beneficial to have high mountains and in general terrain that is not simply flat all throughout. This will add a bit of reality, which in turn adds to the playability.

One solution to this problem is using biomes that have differentiating terrain generation algorithms. The most common biomes that are seen in today's procedurally generated games are: jungle, tundra, desert and woods. [1.] The types of biomes that are usually found in games are shown in image 1. This list contains some of the biomes found in Minecraft.



Icon	Dec	Hex	ID Name	Biome	Color
	0	0	minecraft:ocean	Ocean	000070
	1	1	minecraft:plains	Plains	8DB360
	2	2	minecraft:desert	Desert	FA9418
	3	3	minecraft:extreme_hills	Extreme Hills	606060
	4	4	minecraft:forest	Forest	056621
	5	5	minecraft:taiga	Taiga	0B6659
	6	6	minecraft:swampland	Swampland	07F9B2
	7	7	minecraft:river	River	0000FF
	8	8	minecraft:hell	Hell	FF0000
	9	9	minecraft:sky	The End	8080FF
	10	A	minecraft:frozen_ocean	*FrozenOcean*	9090A0
	11	B	minecraft:frozen_river	FrozenRiver	A0A0FF
	12	C	minecraft:ice_flats	Ice Plains	FFFFFF

Image 1. Biomes found in Minecraft. [20.]

2.2 Environment’s characteristics

Most games that use procedural level generation aim to keep certain characteristics in the generated levels. This helps the players to quickly adjust to the new worlds. Sometimes these characteristics might be certain buildings, certain attributes tied to certain biome types, the main point being that there is always something that is recognizable.

For this project it was decided that it will be the tunnel generation. The tunnel generation should differ from the existing two-dimensional games in tunnel density. Most games have very dense networks, whereas having less tunnels might add to the gameplay value. Having less tunnels adds value to each tunnel, making it more meaningful to find them, meaning it could be made more rewarding. For comparison regarding the density, image 2 shows a cave network from Terraria, another game that has procedurally generated levels.

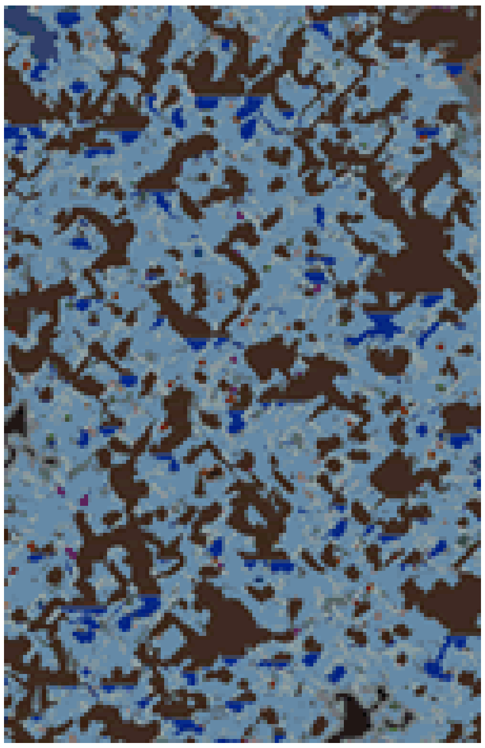


Image 2. Terraria’s cave networks. [21.]

3 Generation algorithms

When choosing the sub-algorithms, it is essential that an initial design is ready. The initial design lays foundation based on which the algorithm will be evaluated.

The term sub-algorithm refers to the pre-existing ontogenetic and teleological standalone algorithms, more specifically the algorithms which were evaluated during the research process of this Bachelor’s Thesis.

It is recommended to study multiple algorithms, how they work and how flexible they are. It is also important to keep in mind that multiple sub-algorithms may be chosen, each to fulfill a different task. In addition, it should be noted that any chosen algorithm can fulfill multiple tasks when used creatively.

There are two main types of procedural algorithms that are used in procedural content generation, ontogenetic and teleological. [4.]

3.1 Ontogenetic algorithms

Ontogenetic algorithms aim to replicate the result of a teleological process. However, ontogenetic algorithms often do so with fewer intermediate steps. [2.] This makes the ontogenetic algorithms more feasible for real-time applications, as they are most often more optimal than their teleological counterparts. The chapters below goes through the internals of two ontogenetic algorithms, Diamond-square and Perlin noise.

3.1.1 Diamond-square algorithm

The diamond-square algorithm is an improvement of another generation algorithm, i.e. midpoint displacement. [11.] The idea of diamond-square algorithm was first introduced by Alain Fournier, Don Fussell and Loren Carpenter. [12.]

Diamond-square improves on the idea of midpoint displacement algorithm by introducing an intermediate step called the diamond step. Both diamond and square steps are explained later in this chapter. The steps behind both algorithms, midpoint displacement and diamond-square are shown in image 3 below.

The algorithm starts by setting an initial value to all four corners of the two-dimensional $2^n + 1$ square array. After the initial setting of the values the actual diamond-square iteration begins and continues until the two-dimensional array has been filled. The diamond and square steps are visualized in the image below.

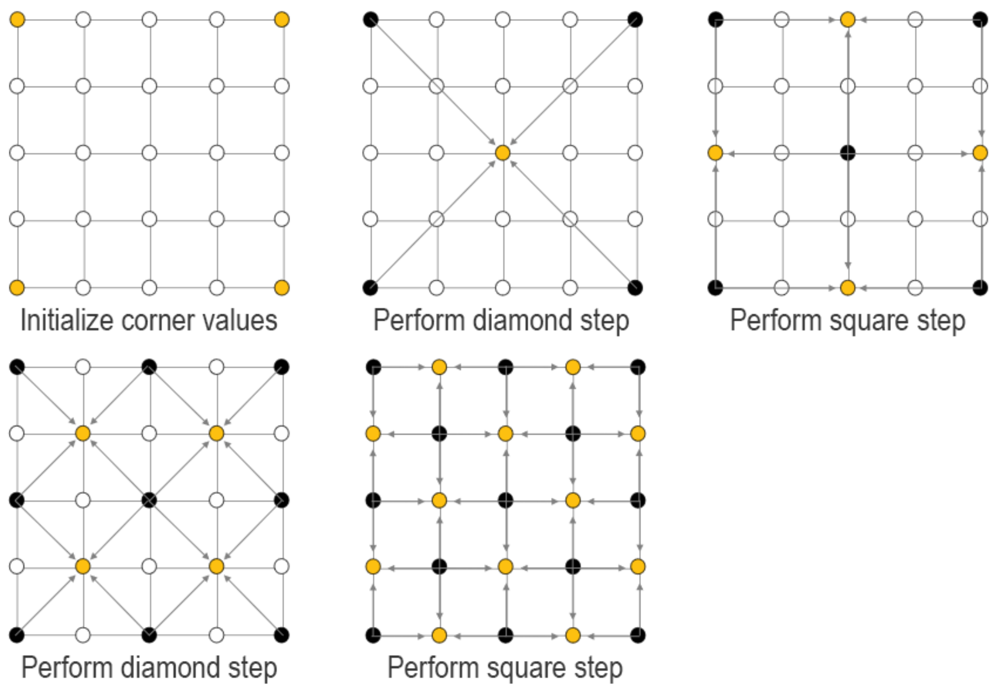


Image 3. Visualization of the diamond and square steps. [8.]

After setting the initial values to all four corners the midpoint of those corners is calculated. The value of that midpoint will be set as the average of the corners plus a random value. This is called the diamond step.

Once the diamond step has been done the midpoint of each diamond in the two-dimensional array will be calculated and its value will be set as the average of the diamond’s corner points plus a random value. This is called the square step.

These two be repeated until all values have been set and it ensures that no value will be set twice. There are some decision choices to be made regarding the algorithm’s implementation, such as how many values to use in calculating the average for points that are close to, or at the border of the two-dimensional array. It could be done via wrapping around and using four values to calculate the average for each point, or just by ignoring the missing values and only using the adjacent values.

Diamond-square algorithm has been described as flawed by Gavis S. P. Miller, because of it producing noticeable creases both vertically and horizontally. [9.]

3.1.2 Perlin noise algorithm

Perlin noise was originally developed by Ken Perlin and was used to generate computer graphics for Disney's computer animated sci-fi motion picture Tron. [13.] Perlin noise's development was inspired by the frustration that was caused by "machine-like" look of computer graphics at that time.

The underlying idea of the algorithm itself is to add layers of noise to the heightmap. Image 4 below shows the effect that octave count has on the generated heightmap. [10.] The effect that octave count has on the generated values is demonstrated in image 4.

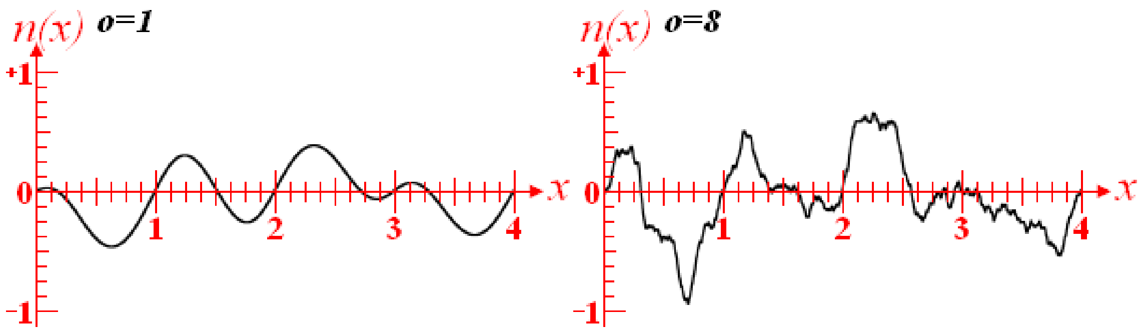


Image 4. Example of values generated by Perlin noise with different number of octaves. [3.]

The gradient noise itself differs from regular random values by having the generated values correlate with each other. This is very powerful and allows the creation of smooth surfaces and meaningful alteration of the surface in general.

Perlin noise is more flexible than most algorithms in the sense that it is not tied to a certain dimension count. Perlin noise can be used in populating heightmaps despite their dimension count.

3.2 Teleological algorithms

The teleological approach is more lifelike, it aims to create a world that could be called an initial draft. Once that initial version has been generated it'll be simulated upon, the results of the simulation should emerge as they would in nature. [4.] In simpler terms, teleological algorithms yield results through iteration. A summary of artificial life algorithms is given in chapter 3.2.1., while chapter 3.2.2. goes over the idea behind the rain drop algorithm.

3.2.1 Artificial life algorithms

Artificial life algorithms are a subgroup of teleological algorithms, meaning that this part of the thesis will not be focusing on a single algorithm but instead the idea behind artificial life algorithms and their attributes.

Artificial life algorithms are very lifelike, they consider the effects of population, reproduction and extinction. These algorithms would be a great tool for creating dynamic environments where the world changes over time. However, they are not the best fit for the world’s initial generation as they a tendency to be slow.

There are multiple attributes that make the usage of artificial life algorithms unintuitive for world generation, one of them being the fact that the world gets fully generated once and large parts of it are being rebuilt multiple times during the iteration process. This makes them considerably slow and it becomes harder to maintain the desired characteristics of the world.

3.2.2 Rain drop algorithm

The rain drop algorithm is used to modify heightmap by simulating the falling of rain drops onto it. Each falling rain drop reduces the height of the point it hits and moves the removed value to either lower or the lowest point in the heightmap. In other words, the algorithm emulates the process of erosion. [14.]

Rain drop is a great alternative to algorithms such as Perlin noise when it comes to surface level generation. The generated values could be modified to leave plateaus to the world. The main problem with the algorithm however is that using it with different amounts of dimensions could prove to be less efficient than it is with Perlin noise.

However, the rain drop algorithm could be used for some specific tasks, such as mountain range generation which would be a task it could perform without issues.

3.3 Ontogenetic vs. Teleological

It is often perceived that ontogenetic algorithms are a better fit for real time applications than teleological algorithms [5.] and this conclusion has been made once again during the research process of this Bachelor’s Thesis.

Teleological algorithms tend to be slow by their nature, it takes multiple iterations to reach a suitable state and in worst case one is never reached. They are however great for iterating on an existing world, making it more dynamic, expanding some of the biomes while shrinking some.

Ontogenetic algorithms on the other hand are great for creating the initial world, they are most of the time very efficient and fast. They are often easily modifiable but shouldn’t necessarily be used to modify the world after the initial generation process has been finished.

Based on the results of the research process it can be concluded that both algorithm types have their use cases. This does not mean that they couldn’t be used to fulfill each other’s use cases, it is more a question of suitability than anything else.

4 Algorithm design

This chapter goes over the design of the developed algorithm and based on which factors the decisions were made. When designing such algorithms, it is important to know what the desired output of the algorithm is and what sub-algorithms are to be used. When both of those are factored in it can then be decided how many dimensions are to be used in the value generation.

For example, Perlin noise can be used to populate either, one or two-dimensional value maps. It can be used to populate value maps with more than two dimensions but for this Bachelor’s Thesis that is not necessary.

4.1 Algorithm type

For this Bachelor’s Thesis it was decided that ontogenetic algorithms would be a better option due to them being more geared towards real-time applications, such as games. [5.] This does not mean that teleological algorithms cannot be used for real-time applications. The decision was mainly affected by the fact that the values generated via ontogenetic algorithms were a more viable option in terms of optimization.

4.2 Order of generation

The order in which the parts of the environment are generated can drastically change the output. For example, if tunnels were to be generated prior to biomes, the biomes would end up modifying the biome within the tunnels. If tunnels were being generated after biomes however, it could be made so that they will default to a certain biome, ensuring that the biome remains the same throughout the tunnel.

The way in which order of generation affects the outcome might be difficult to grasp. It’s one of those things where it is recommended to seek out the desired result by trial and error. It would be beneficial to determine this during the initial design phase but it is very abstract and difficult to do so.

For this Bachelor’s Thesis it was decided that the tunnel generation will be done after everything else. This was decided to ensure that the caves have the one and same biome all throughout the world. All decisions related to the order of generation were made by testing them first. They were not defined as a part of the initial design.

4.3 Generation modifiability

For the generated levels to have noticeable differences between each other the algorithm should use multiple generated value maps that control different parts of the level.

The reason for having multiple generated value maps is that the differences in a single map’s values between each iteration may not differ that much. Adding a few more maps that control different areas of the level however makes those little differences add up while that should not affect the characteristics of the game because the maps are still used the same way as they were before.

There are workarounds for this, like using one large map for the generated values and dedicating certain areas for certain entities. Even with this in mind it still is advisable to keep the generated values in separate maps as this allows the easy modification of the variables used in the value generation.

4.4 Expandability

Expandability is considered to mean the flexibility of the technical implementation. It is essential to keep the internals of the system as flexible as possible. The flexibility makes the development as well as further updating more feasible.

A more tangible example of this is taking into consideration how effortless the process of adding new attributes should be? For example, if a new biome type is to be added or one is to be removed, how many lines of code need to be modified, or is it even possible with the developed system?

For this Bachelor’s Thesis it was decided during the initial design that all the attributes of this nature shall be stored in enumerated types or other constant variables. The solution itself is very simple, as it should be and allows the ease of use when removing or adding attributes, such as biomes.

5 Algorithm implementation

The developed algorithm has many attributes that make the world generation easily modifiable, these attributes are inspected more closely in chapter 5.2.

5.1 Chosen sub-algorithms

The algorithm relies on a single sub-algorithm, Perlin noise. It is used to generate multiple heightmaps and all those maps are being utilized differently. For example, surface level has been created using the generated values directly, although ignoring some of them to create plateaus.

The heightmaps are being populated using a single algorithm, Perlin noise. Perlin noise was chosen as the only algorithm for value generation due to its flexibility, it is usable even in one-dimensional value generation. The generated values are however being used in multiple ways. For example, the tunnels are being generated by Perlin worms.

5.1.1 Plateau generation

The plateaus were made possible by having minimum value that was read from the Perlin noise algorithm. In other words, when Perlin noise would return -0.75 it would be interpreted as -0.5, meaning that the range between -0.5 and -1 would share the height, leaving plateaus all around the map. The mountains on the other hand exist because the values are not being reinterpreted in any way on the high end of the value range.

5.1.2 Pocket generation

The algorithm generates small pockets as shown in image 5. In the current implementation these pockets are simply holes surrounded by a specific biome. The algorithm generates a certain amount of them by using a dedicated heightmap. The two-dimensional heightmap is considered to reflect the world’s coordinates and is then pseudorandom points are being checked for values. Each point that has a value greater than the set threshold will be made into a pocket with set diameter. The algorithm makes no checks regarding the terrain’s current condition, meaning that theoretically it would possible for all the pockets to be created at the exact same point. The pseudorandom coordinate generation does have a set limitation, the center point of any pocket cannot be above the surface level.

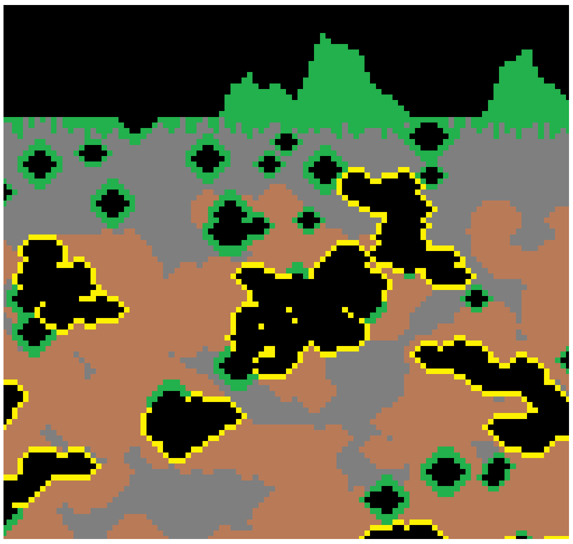


Image 5. Example of the generated pockets.

Pockets are the small green holes with green surroundings and as the image shows some of them are very close to each other because there are no checks regarding this.

5.1.3 Tunnel generation

The tunnel generation uses values generated by Perlin noise, as well as everything else in this algorithm. The method used to create the tunnels from the values is called “Perlin worms”.

Perlin worms takes a certain amount of values from the value map populated by Perlin noise, the values are usually taken along either X or Y-axis. The spacing between each taken value should be noticeable as Perlin noise would otherwise return values that are too close to each other. Each taken value represents a direction. Image 6 below clarifies the usage of the picked values.

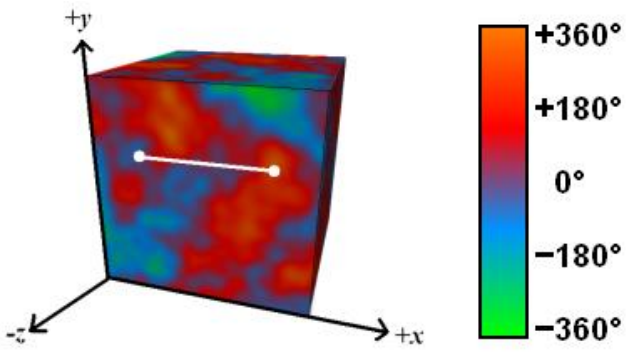


Image 6. Visualization of values picked for Perlin worms. [7.]

The actual values used by the algorithm are as follows:

- > 0.25, Left
- > 0, Right
- > -0.25, Up
- < -0.25, Down

The values are checked in the listed order. The greatest factor in how the Perlin worms behaves was the gap between each picked value. This part of the algorithm went through a lot of iterations.

5.2 Generation parameters

The algorithm has plenty of modifiable attributes that have direct effect on the result. These attributes are listed in the same order in which they are being used by the algorithm. In addition to listing the attributes this subchapter mentions the attributes' currently set values.

5.2.1 Surface generation attributes

Surface level generation uses two modifiable attributes, the first one of them being the octave count used by Perlin noise in the heightmap population. The effect that octave count has on the generated values is viewed more closely in chapter 3.1.2. In addition, the surface level generation has control over the plateaus. Perlin noise produces values between -1 and 1 by default and the plateaus are made by the low end of that range. The width of the ignored range is modifiable and has direct effect on the size of the plateaus.

In the current implementation of the algorithm all the negative values are being ignored when generating the surface level. This leads to a surface that has very wide plateaus and as such limits the size of the mountain ranges.

5.2.2 Biome and cave generation parameters

Biome and cave generation use the same style of generation and as such, also share the generation attributes. The first attribute is the octave count, as it was with the surface level's generation.

The next attribute that the algorithm sets is the number of worms to be generated. After that the limitations for the starting points are set, for example cave generation's starting point is set to never be above surface level. There are three more variables regarding tunnels that can be set to have different values for each one, the tunnel's length, each point's diameter and the distance between each point. The length is the amount of points the worm consists of. The diameter is used when selecting tiles around the center point and modifying them, either removing or changing something about them and finally the distance between each point which is currently set to be equal to the diameter.

The number of worms being generated is the same for biomes and regular caves, however, they have a significant difference in each point's diameter. The diameter used in biome generation is close to three times as large as the one used in cave generation, this leads to it being very unlikely that there would be individual parts of the biome that aren't connected to the rest.

To demonstrate the significance of these parameters below are two images, image 7 and image 8. The worlds shown in the images have different amount of generated worms for both, caves and biomes. The pocket generation for both of these worlds remains the same.

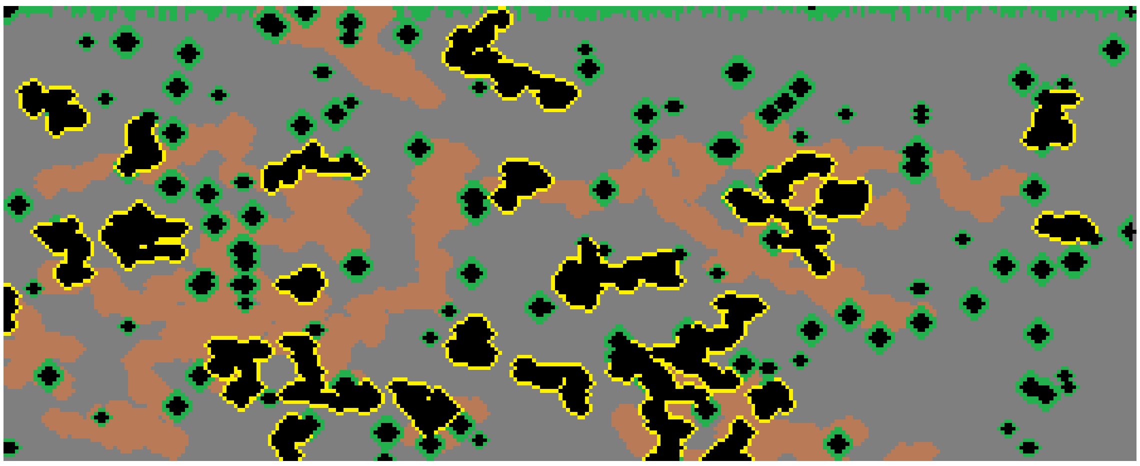


Image 7. An example of a world that was generated with high tunnel count (15-30).

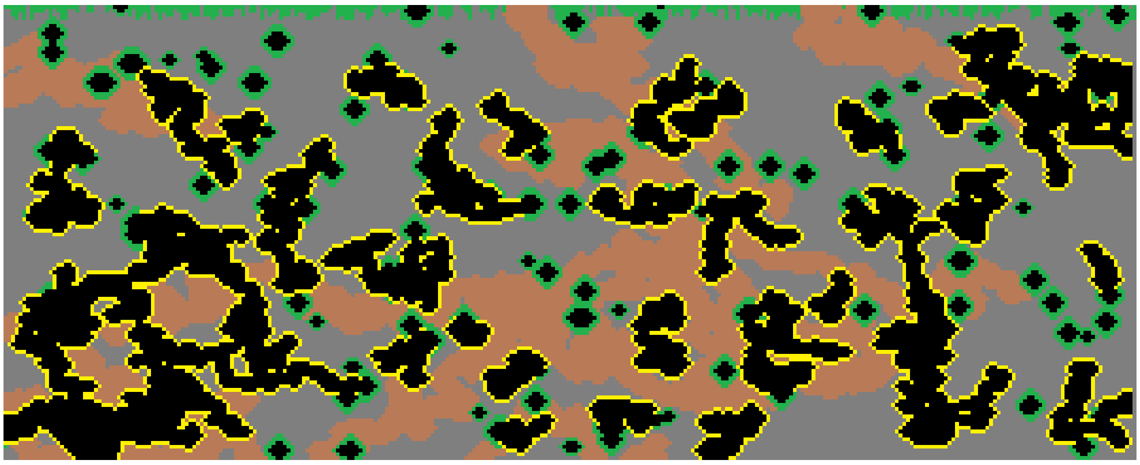


Image 8. An example of a world that was generated with high tunnel count (75-100).

As the images above show, there is a significant difference in the amount of explorable content between these two worlds.

5.2.3 Pocket generation parameters

Pocket generation shares a lot of attributes with cave and biome generation. It is basically the same but without the worm effect. There is the octave count as with everything else. The coordinate limits, which in the current version share the cave generation's limitations, starting points are to be below the surface level.

In addition, there are two attributes, the number of pockets being generated and the width of each pocket. The width of the pockets is guaranteed to be within a set range, this range can be modified for direct effects on the results. In the current implementation this range is very small.

6 Internals

This chapter focuses on the used technologies, libraries and tools. For this Bachelor's Thesis it was decided that modern C++ will be used. The term modern C++ refers to C++11 and newer.

6.1 Engine

The game engine that this Thesis uses has been custom built using object-oriented C++'s functionalities. The engine itself is built upon two third party libraries, Simple DirectMedia Layer and Box2D.

Simple DirectMedia Layer is a C-based library and as such required some interfaces to be built for it to be convenient to use in a modern C++ environment. Box2D on the other hand is based on C++. Even though Box2D is based on C++ it did take some interface re-designing to get it to work well together with the existing engine's structure and Simple DirectMedia Layer.

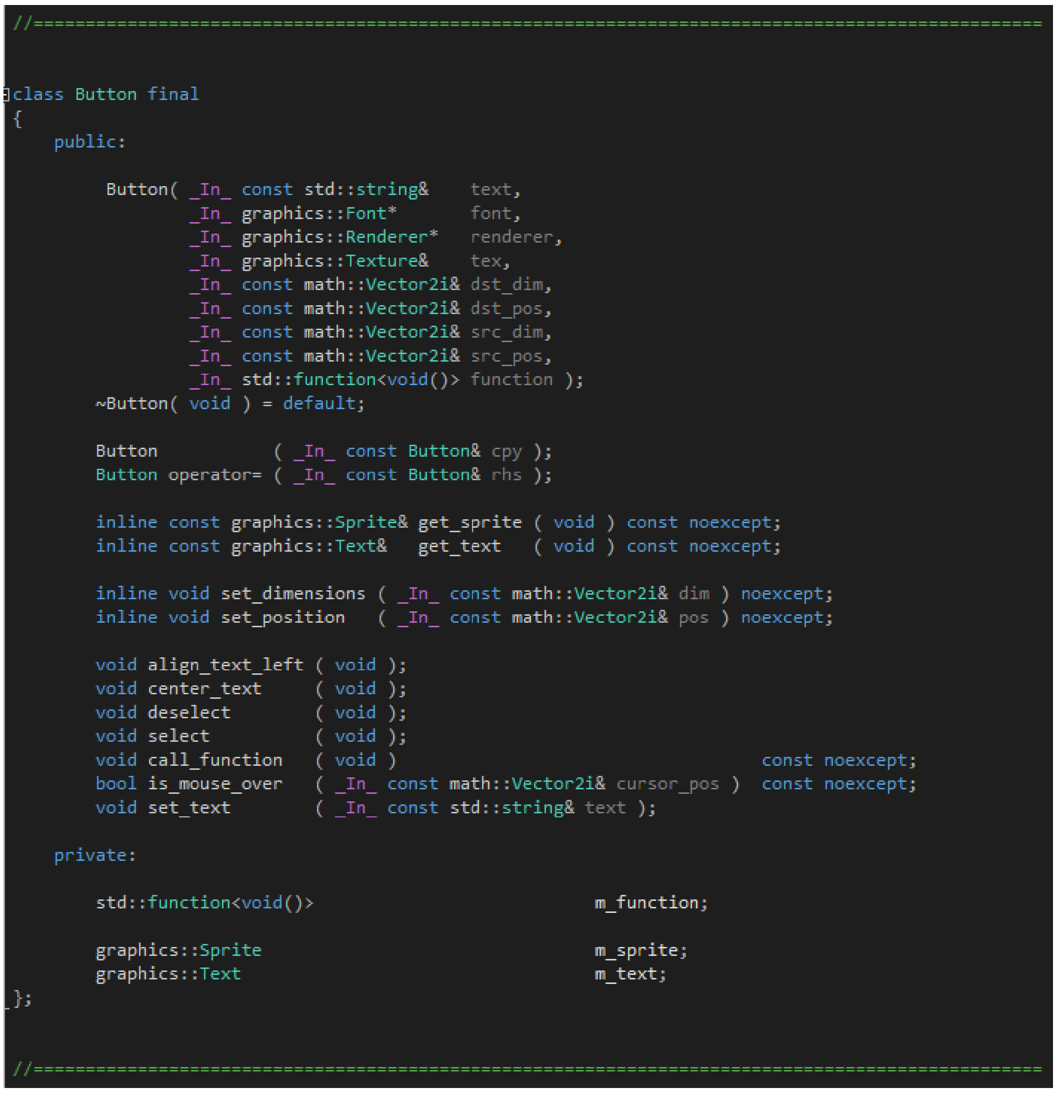
6.2 Object pooling

The tiles are handled by a method called object pooling. Object pooling itself refers to a method where upon launching the game more objects of certain type are being created than what's needed at that moment. A method like this makes the game's initial launch more time consuming but reduces the need for runtime memory allocations.

The engine supports this feature in the form of a base class which only contains a flag and an accessor and a mutator for that flag. The flag's sole purpose is keep track of which objects are active and which are not. The engine will ignore the physics calculations and rendering of inactive objects.

6.3 Programming conventions

For the Thesis, it was decided that set conventions are to be followed for the entirety of the project. This led to the code being easier to read, modify and extend upon. Every class, static function and variable are inside a descriptive namespace. In addition, all functions are separated from each other by a clear 100 characters long separator line and all parameters of all the functions have been annotated. Image 9 shows an example of a header file that has been implemented by following the convention.



```
//=============================================================================================

class Button final
{
    public:

         Button( _In_ const std::string&    text,
                 _In_ graphics::Font*       font,
                 _In_ graphics::Renderer*   renderer,
                 _In_ graphics::Texture&    tex,
                 _In_ const math::Vector2i& dst_dim,
                 _In_ const math::Vector2i& dst_pos,
                 _In_ const math::Vector2i& src_dim,
                 _In_ const math::Vector2i& src_pos,
                 _In_ std::function<void()> function );
        ~Button( void ) = default;

        Button           ( _In_ const Button& cpy );
        Button operator= ( _In_ const Button& rhs );

        inline const graphics::Sprite& get_sprite ( void ) const noexcept;
        inline const graphics::Text&   get_text   ( void ) const noexcept;

        inline void set_dimensions ( _In_ const math::Vector2i& dim ) noexcept;
        inline void set_position   ( _In_ const math::Vector2i& pos ) noexcept;

        void align_text_left ( void );
        void center_text     ( void );
        void deselect        ( void );
        void select          ( void );
        void call_function   ( void )                                  const noexcept;
        bool is_mouse_over   ( _In_ const math::Vector2i& cursor_pos ) const noexcept;
        void set_text        ( _In_ const std::string& text );

    private:

        std::function<void()>                          m_function;

        graphics::Sprite                               m_sprite;
        graphics::Text                                 m_text;
};

//=============================================================================================
```

Image 9. Example of a class’ header file that follows the set convention.

7 Evaluation

This chapter goes over the strengths and faults of the developed algorithm. The strengths only include functionality that the algorithm has in its current form. The fault inspection takes a wider look at the algorithm ignoring the scope of the project and the limitations that came with it.

7.1 Strengths

This subchapter compares the developed algorithm to the goals that were originally set for it. In addition, this chapter goes over other features of the algorithm, the ones that were not originally planned.

7.1.1 Initial goals

The developed algorithm meets all the goals that were initially set. The algorithm can generate cave networks of varying sizes and allows the modification of tunnel length, width as well as the number of tunnels in total.

The surface level generation meets the original goals as well, there are clear plateaus and a few mountains, although the mountain ranges could use more modifiability.

7.1.2 Extendibility

The algorithm and its structure are very easy to modify and extend upon. The algorithm was designed in a way where all generation steps, like surface and tunnels are completely unrelated to each other. This makes it easy to add more steps, modify single steps without affecting the rest of them and overall makes the experimentation enjoyable.

7.2 Faults

This subchapter describes some of the faults as well as possible improvements for the algorithm. It should be noted that most of the items listed here were not realistically implementable due to time restrictions.

7.2.1 Biome generation

The biome generation could have been improved in a variety of ways, starting with the borders of each biome. The biome borders are very flat, changing from one to another without any kind of blending. One solution to this issue could have been a border where the used biome would be chosen based on pseudorandom values. Another, more feasible solution would have been using linear interpolation, making the transition smoother between each biome type. The idea of this methodology is explained in image 10 below.

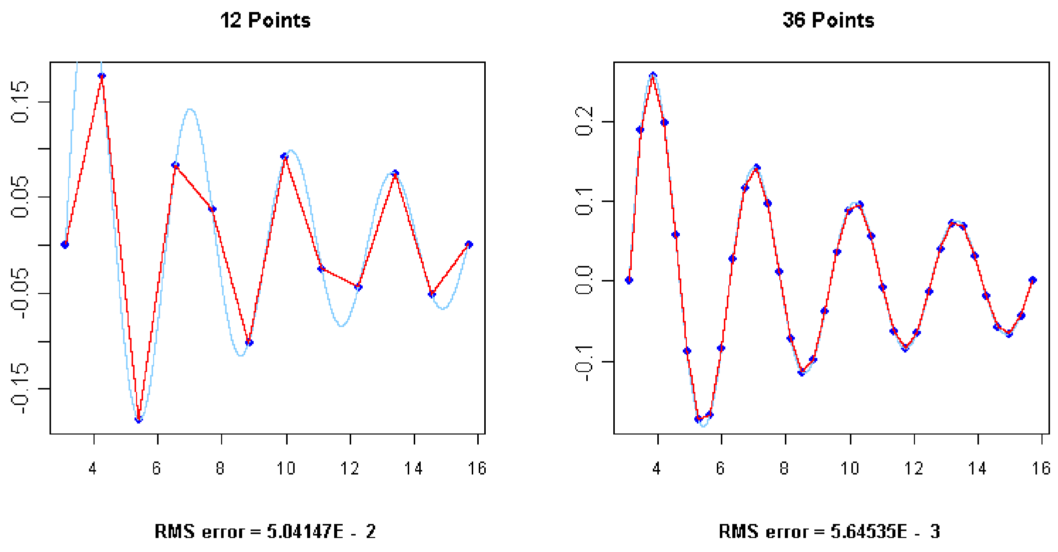


Image 10. Visualization of linear interpolation that could be used in merging the biomes. [22.]

In addition, it would add immense value to the algorithm to have different surface generation based on biomes. For example, desert biome's surface level could have used sine waves to generate the height map instead of the Perlin noise that is used in the current version. This biome-specific generation could have been applied to the algorithm.

One additional attribute that should have been taken into consideration during the initial design is that the biome generation should differ based on the current height level. This would add to the gameplay value and is even realistic, it is very likely to have completely different biomes deep underground than on the surface.

7.2.2 Surface generation

The surface generation was very minimal, it only uses a single value map that has been populated using Perlin noise. In addition, the fact that plateaus are implemented by ignoring a contiguous part of the value range simplified the surface generation even further.

Overall, the surface level could use more shapes as it now consists of flat plateaus and high mountains. A great improvement would have been having round hills. Round hills could be implemented by having a specific value map for them, a more optimal solution would have been using Bézier curves to create such hills. Image 11 shows an example of a Bézier curve.

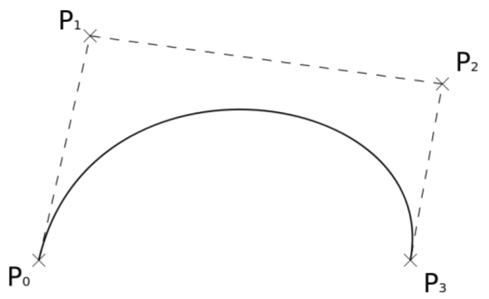


Image 11. Cubic Bézier curve with four control points. [6.]

Bézier curves could have been dedicated to such hill generation and they would have had much less overhead than a dedicated value map would.

7.2.3 Plateaus

The plateau generation could have been improved by having them at multiple places along the Y-axis. In the current version all the plateaus are on the same level, a level which could be considered as the surface's base level.

The current version simply ignores parts of the value range that Perlin noise uses and in doing so creating height levels that are shared between certain value ranges. An optimal solution that would have greatly affected the outcome is having multiple smaller number ranges act as plateaus.

The number of plateaus and their size could have been controlled simply by changing the range which is considered as a plateau. For example, if wider plateaus were desired the range would have to be increased. If, however, more plateaus were desired the range could be split into smaller groups. For example, the following groups could be considered as plateaus:

- -0.25 – 0
- 0.35 – 0.5
- 0.7 – 0.8

Each of these value ranges could have used their own height level and added a lot of variety to the algorithm.

7.2.4 Mountain ranges

The mountain ranges could be their own biome; however, they are often found in multiple biomes and as such them not being one is not considered to be a fault on its own. The fault lies in the fact that they don’t have their own heightmaps. Mountain ranges would stand to gain great benefit from having a dedicated heightmap with high octave count and a sub-algorithm that would take the full range of values into account.

This improvement would make the generation significantly slower as it would be beneficial to have a dedicated heightmap for each biome’s mountain range. However, most applications that have a generation algorithm of this type only do it once for each world and the worlds are being generated very rarely. This mean that while it is not desirable to have a heavy algorithm it is also not completely unheard of.

Also, as mentioned in chapter 3.2.2 the rain drop algorithm would have been a great fit for this application. Rain drop algorithm, when used together with Perlin noise would have worked well, making the mountain ranges differ from the rest of the surface generation, making them even more recognizable.

7.2.5 Variety

The current algorithm falls short when it comes to the level of variety between each iteration. The plateau generation’s implementation results in a great loss of variety, by ignoring all negative values most of the surface level will remain flat as shown in image 12 below, and as such instead of adding to the gameplay value it takes from it instead.

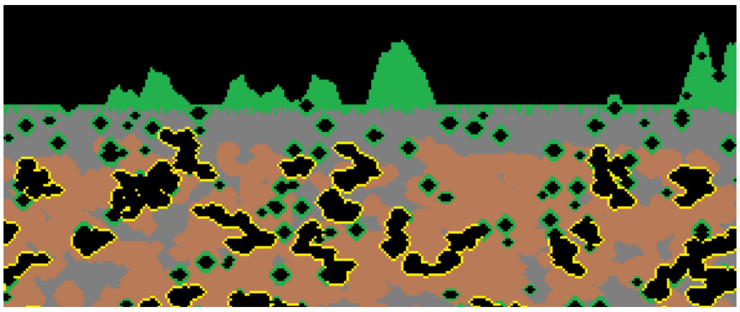


Image 12. Example of the resulting surface level's flatness.

In addition, both cave and pocket generation suffer from the issue that no system exists to ensure that they are not created on top of each other. While it is acceptable and in fact beneficial that they cross each other their starting points themselves should never overlap.

7.2.6 Validation

When considering usual procedural generation that is going into production validation plays a big part in it. There are two possible solutions to this, either the algorithm itself is made in a way that ensures it will never produce results that are not fit for playing or there is a validation algorithm.

For the algorithm that has been developed here the first approach would be more beneficial and implementable. The only parts of the algorithm that would require such verification are the cave and pocket generations. The algorithm should ensure that there is at least a certain distance between the existing caves and pockets and the starting point of the one that is being created.

8 Procedural generation in games

Procedural generation has always aimed to solve some issues that arose with early computers and gaming consoles. Any given device has a set amount of resources, memory limitations and because of that an infinite amount of textures or worlds cannot be stored.

Using procedural generation helps in reducing the size of the shipped title, as generated content does not have to be stored in the shipped unit.

The overall benefit of procedural generation is that the shipped game takes less space, provides infinite amounts of content and takes away from the predictability, keeping the gamers more alert.

8.1 Texture generation

Although memory itself is not as big of an issue as it used to be some game developers want to go the extra mile to ensure that the game itself remains as small as possible. An example of such game is RoboBlitz, which was released back in 2006. [18.]

RoboBlitz uses procedural generation for its textures to ensure the small size, it was beneficial for the game to be small at the time since it was one of the games that were being distributed through the online systems. The game itself ended up being less than 50 megabytes on Xbox Live. [18.] Image 13 shows how the game RoboBlitz ended up looking with its procedurally generated textures.

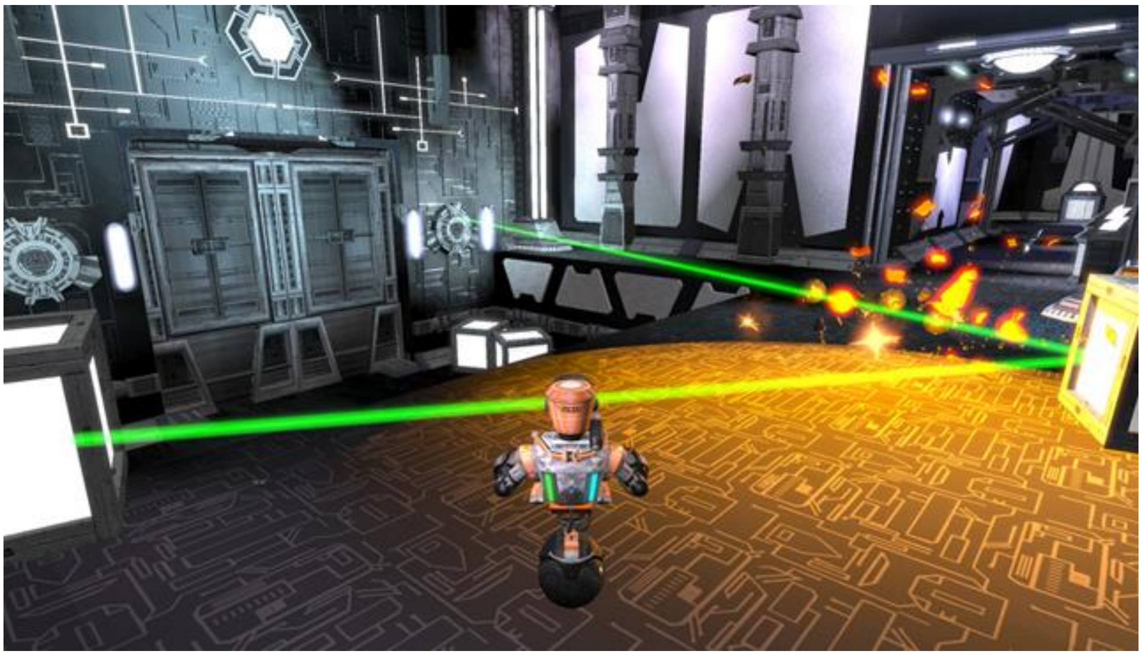


Image 13. Gameplay picture of the game, RoboBlitz. [19.]

RoboBlitz used a middleware tool for the storing of its procedurally generated textures. The tool was developed by Allegorithmic. [18.] Allegorithmic aims to create procedural generation more user-friendly through their toolsets that they offer for developers.

8.2 Other use cases

Procedural generation is being used in more complicated tasks than just texture and world generation, it is being utilized in the creation process of three-dimensional models.

One of the more recent games that used procedural generation in more than just world generation was a game called "No Man's Sky". [15.] Image 14 below demonstrates what the generated creatures look like before the body parts have been chosen by the algorithm.

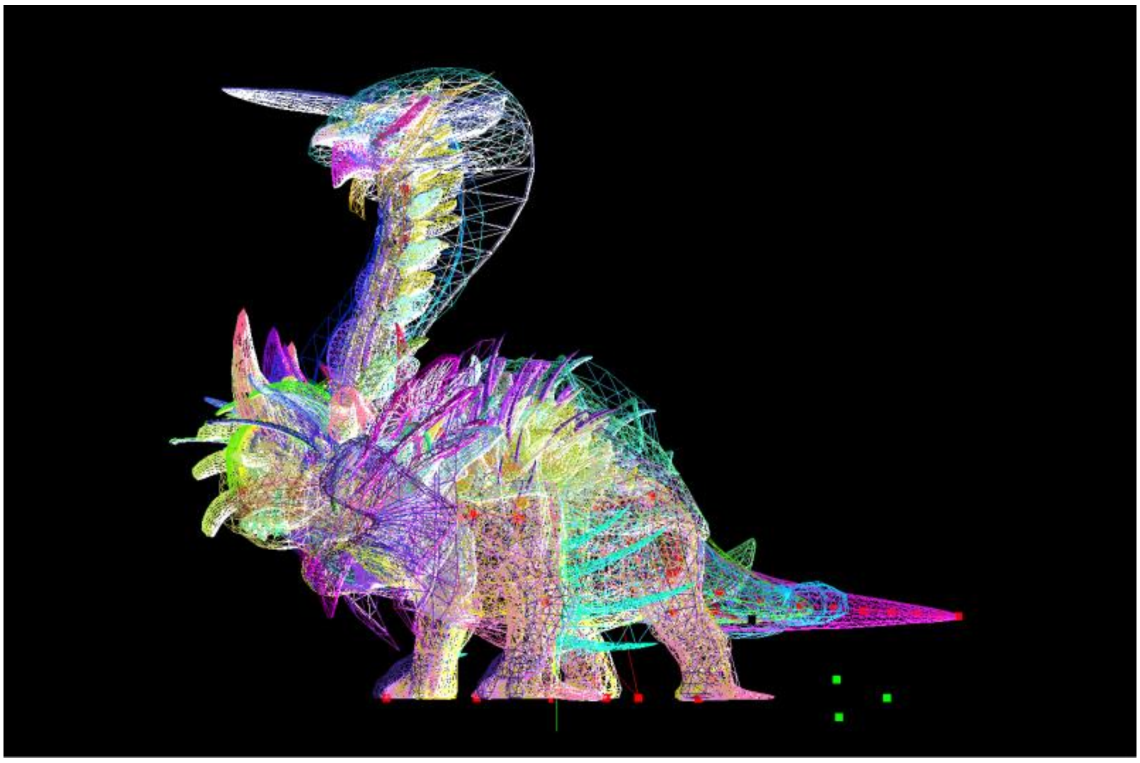


Image 14. Image of three-dimensional model from No Man's Sky. The model consists of all the possible body parts a creature can have. [16.]

The image might be hard to understand at first, but it contains all the body parts that are in that game and they are being combined by the algorithm to create the creatures that roam around the game world. Image 15 below shows a collection of creatures created by the algorithm.

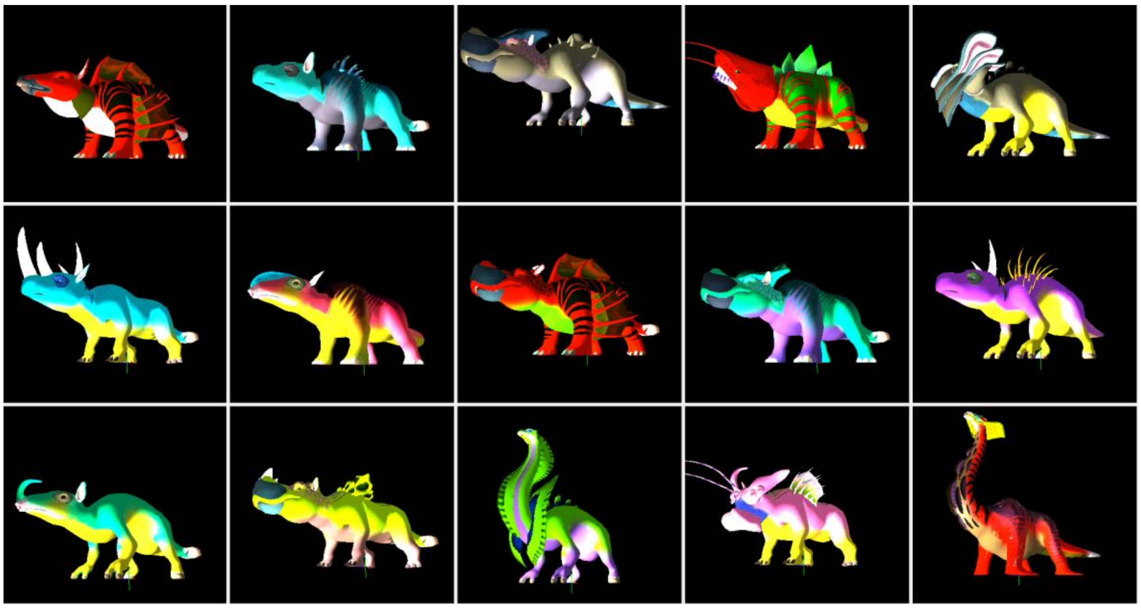


Image 15. Creatures created by the procedural generation algorithm of No Man’s Sky. [17.]

The fifteen animals seen below have been combined from all the possible body parts by the game’s own procedural algorithm.

8.3 Future of procedural generation

The hardware that is being used by gamers has improved a lot over the past three decades and the tools used in texture creation have advanced enough to make very fast-paced creation of textures possible without such algorithms. Meanwhile, the development of said algorithms has remained slow and because of this it is not being utilized in texture generation as much anymore. There are exceptions to this. Games that require immense amounts of textures are still better off using procedurally generated textures than having artists create them by hand.

Currently procedural generation is mostly used in world generation when it comes to games. There are also games that use it in creating three-dimensional models but due to the complexity of such algorithms it is most often more sensible to have dedicated artists creating such models. There are however exceptions to this, just like the texture generation, if a game requires an immense amount of such models it becomes more beneficial to develop an algorithm for it.

Overall it can be concluded that the need for procedural generation is not necessarily as big as it has been in the past when memory was more of an issue. Procedural has however remained to be the main selling point of some games because of its theoretical ability to provide an infinite amount of content.

9 Conclusion

The original goal of this Bachelor’s Thesis was to implement a procedural algorithm for creating worlds that meet certain requirements.

During the initial design process both ontogenetic and teleological algorithms were considered, and it was decided that ontogenetic algorithms would be a better fit for the algorithm because of their attributes. However, it was later discovered that combining both of those algorithm groups would have been beneficial for the algorithm.

The algorithm is far from perfect; however, it meets most of the requirements that were initially set for it and much was learned during the development process. Most of the improvements were left undone because of time restrictions and the fact that it was decided that the algorithm will be developed according to the initial design choices.

References

1 Minecraft’s biomes. Referenced 20.10.2017.
https://minecraft.gamepedia.com/Biome

2 Teleological vs. Ontogenetic. Referenced 25.10.2017.
http://pcg.wikidot.com/pcg-algorithm:teleological-vs-ontogenetic

3 Modifying number of octaves when using Perlin noise. Referenced 21.10.2017.
http://libnoise.sourceforge.net/glossary/index.html#octave

4 Teleological vs. Ontogenetic. Referenced 22.10.2017.
https://www.gamasutra.com/view/feature/130071/random_scattering_creating_.php?page=2

5 “Shattering Reality”, Game Developer, August 2006. Referenced 23.10.2017.
http://twvideo01.ubm-us.net/o1/vault/GD_Mag_Archives/GDM_August_2006.pdf

6 Cubic Bézier curve with four control points. Referenced 24.10.2017.
https://en.wikipedia.org/wiki/B%C3%A9zier_curve#/media/File:Bezier_curve.svg

7 Visualization of values picked for Perlin worms. Referenced 24.10.2017.
http://libnoise.sourceforge.net/examples/worms/images/cube.png

8 Visualization of the Diamond-square algorithm. Referenced 25.10.2017.
https://en.wikipedia.org/wiki/Diamond-square_algorithm#/media/File:Diamond_Square.svg

9 ACM SIGGRAPH Computer Graphics, August 1986. Referenced 25.10.2017.
https://dl.acm.org/citation.cfm?doid=15886.15890

10 Perlin Noise, Octaves. Referenced 29.10.2017.
http://pcg.wikidot.com/pcg-algorithm:perlin-noise

11 Diamond-square algorithm’s origins. Referenced 29.10.2017.
http://pcg.wikidot.com/pcg-algorithm:diamond-square-algorithm

12 SIGGRAPH Communications of the ACM, June 1982. Referenced 29.10.2017
https://dl.acm.org/citation.cfm?doid=358523.358553

13 Perlin noise, Wikipedia. Referenced 29.10.2017.
https://en.wikipedia.org/wiki/Perlin_noise

14 Rain drop algorithm. Referenced 29.10.2017.
http://pcg.wikidot.com/pcg-algorithm:rain-drop-algorithm

Metropolia

15 Procedural generation in No Man’s Sky. Referenced 29.10.2017.
https://kotaku.com/a-look-at-how-no-mans-skys-procedural-generation-works-1787928446

16 Creature model from No Man’s Sky. Referenced 29.10.2017.
https://i.kinja-img.com/gawker-media/image/upload/s--puU1RG0q--/c_scale,fl_progressive,q_80,w_800/hgxydd9pfcvsvqb4n2aj.png

17 Creatures generated by the algorithm of No Man’s Sky. Referenced 29.10.2017.
https://i.kinja-img.com/gawker-media/image/upload/s--l5GxP2PP--/c_scale,fl_progressive,q_80,w_800/fidynxo2teot0jb6hapz.png

18 RoboBlitz, Wikipedia. Referenced 29.10.2017.
https://en.wikipedia.org/wiki/RoboBlitz

19 Gameplay image from RoboBlitz. Referenced 29.10.2017.
https://en.wikipedia.org/wiki/RoboBlitz#/media/File:RoboBlitz_-_Screenshot_01.jpg

20 Minecraft’s biomes. Referenced 29.10.2017.
https://minecraft.gamepedia.com/Biome/ID

21 Terraria’s caverns. Referenced 29.10.2017.
https://terraria.gamepedia.com/File:Cavern_map.png

22 Visualization of linear interpolation.
http://www.codecogs.com/users/1/linear-378.png