UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Landscape generation using procedural generation techniques

Author:
Vladyslav MELNYCHUK

Supervisor:
Oles DOBOSEVYCH

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences
Faculty of Applied Sciences

Lviv 2020

Declaration of Authorship

I, Vladyslav MELNYCHUK, declare that this thesis titled, “Landscape generation using procedural generation techniques” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

"The trees and the grasses and all things growing or living in the land belong each to themselves."

J.R.R. Tolkien

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Landscape generation using procedural generation techniques

by Vladyslav MELNYCHUK

Abstract

This work is about procedural content generation and its applications in video games. Generating a landscape is one of the ways to use procedural generation in games. The goal of this work is to test different techniques and approaches to develop a foundation for a game that is capable of creating beautiful, realistically looking landscapes. The predefined rules must fully control the generation process. The user input for the generation will be limited to the seed that defines the initial state of the generation, the parameters that control the generation, textures and 3d models. The rest of the work is automated and requires no human interaction. The results of this work can be used as a foundation for different types of games.

Acknowledgements

I want to say thank you to my family, university staff and teachers, and everyone who helped me along this long way.

Contents

List of Figures

List of Abbreviations

PCG	**P**rocedural **C**ontent **G**eneration
SIGGRAPH	**S**pecial **I**nterest **G**roup on Computer **GRAPH**ics and Interactive Techniques
LERP	**L**iner Int**erp**olation
2D	**2 D**dimentional
3D	**3 D**dimentional
VFX	**V**irtual **E**ffe**cts**
LOD	**L**evel **O**f **D**etail
GPU	**G**raphics **P**rocessing **U**nit
CA	**C**ellular **A**utomata

Chapter 1

Introduction

1.1 Procedural generation in game development.

Procedural generation is a powerful tool that is widely used in video game development. PCG can be used for small-scale endless level generation in 2d hyper-casual mobile games, and in complex 3d open-world games. It allows developers to create unique and realistic worlds by defining a set of rules that generate landscapes, textures, characters, buildings, etc. This type of generation also allows games to have infinitely large worlds, as there is no need to store all the data. When the user turns the game on again, everything can just be regenerated. But random generation does not necessarily mean that it is procedural, as there is a difference between random and procedural generation. Random generation uses randomness to generate content from already created materials. It could be textures, 3d models, or simply characteristics of an in-game item. But they are always picked from a predefined set of values. Procedural generation, however, uses preset rules to generate unique, new content. In this work, procedural generation will be used.

1.2 Relevance

One of the most popular games in history is Minecraft, with over 180 million copies sold [*2011's Minecraft becomes the second best-selling game of August 2019*]. The game uses advanced procedural generation algorithms to create entire infinite worlds with landscapes, oceans, caves, towns, and wildlife. It uses seeds to control the rules of generation, allowing users to create 2^{64} unique 3d worlds, that are incredibly realistic, despite being made entirely of blocks and having pixelated graphics. Another popular video game called No Man's Sky also uses PCG, but its developers take the generation on another level. It uses a unique seed to generate an entire galaxy with 2^{64} planets in it to explore. But unlike Minecraft, not only the terrain is procedurally generated, but plants and animals too. Each planet is unique, just like in real life.

PCG can also be a great way to make your game stand out. Video game industry generates US$134.9 billion worldwide [*2018 Global Games Market Report*]. A lot of that money comes from the games developed by big companies [*Top 25 Public Companies by Game Revenues*]. Using PCG in your video game can make players have more interest in it, as generated content is not bound by human imagination or creativity. If fact, the first versions of Minecraft were developed by just one person, and one of the reasons that the game became so popular is that the content was not repeatable and unique.

1.3 Project goal

The goal of this project is to create a realistic landscape with mountains, valleys, lakes, and then fill it up with trees, to make it look like a real mountain range.

To achieve this, a 3d mesh will be generated and reshaped, so it looks like terrain. Next, this mesh will be coloured and textured. The textures will give it realistic looks, adding elements like water, sand, gravel, rocks, and grass.

Then the textured landscape will be filled with trees, rocks, and other decorations that exist in real forests.

For all these steps, different algorithms, techniques and approaches will be used.The entire process of creating the landscape will be procedural and it will follow preset rules. However, the trees and other plants or decorations used for this project will be picked from a preset collection, so they will not be generated, but their placement will be procedural.

Chapter 2

Background information

2.1 Procedural Content Generation

Procedural content generation (PCG) is an automated algorithmic creation of content with limited, or no user input [Julian Togelius, 2016]. PCG can be used to generate data, art, video game content, textures, 3d models, etc. This work focuses on using PCG in video games. In the context of video games, the word content means everything that makes up the game: levels, graphics, in-game items, game rules, music, or maps. [Julian Togelius, 2016] Game developers started using PCG in video games early in the history of video games. Back in the 1980s memory was a huge issue, as computers and game consoles had very little memory. This meant that making larger, or more complex games was challenging as developers had to reuse all game assets continuously, and optimise memory usage. Instead of saving assets to memory, some games started using PCG. This allowed defining rules that generate the game contents, so not everything had to be directly saved to the memory. One of the most famous early games that used PCG is called Rogue [DAHLSKOG, 2016]. It was developed for UNIX systems (later ported for other platforms and consoles) and generated a maze-like dungeon. It allowed to have a lot of levels, with big dungeons, left for the players to explore. The game even gave birth to an entirely new genre of video games.

Nowadays, memory is not a big issue for computers, and their computational power has increased. But the need for PCG in video games has not decreased. Developers use more advanced PCG techniques and algorithms to generate more exceptional content. PCG is also great for player retention and the replayability of the game. Diablo III uses generation to create different dungeons. Having this feature ensures that players will play the game more, as there is always something new to explore. The game Spore is capable of generating planets, structures, and even animated creatures, who have their own cultures and technologies [DAHLSKOG, 2016]. The list of games that use PCG is getting bigger and bigger, and developers find new ways to use PCG for their games.

2.2 Unity3d

Unity3d is a cross-platform game engine [*Unity User Manual*]. It offers a lot of useful tools and assets for game development. It takes care of rendering and provides a high-level API for video games development. The engine also offers libraries with optimised implementations of popular algorithms, tools for development, and an extremely intuitive graphical user interface. Unity3d also provides functionality to create and modify meshes, which will be useful for this work. A mesh data structure is a representation that organises the vertex, edge, and face data so that these queries can be done easily [Gortler, 2012a]. In Unity, a mesh consists of triangles. It also offers a Mesh component that can create meshes. It is done by passing it an array of vertices, UV's and triangles. After this, the engine takes care of creating the mesh, drawing it to the screen and calculating its normals.

Using Unity3d also makes it is possible to create custom shaders. Shaders are a special-purpose program that is written in C-like special language [Gortler, 2012b]. Unity's shader program uses a variant of High-Level Shading Language (HLSL). HLSL is sometimes called CG (C for Graphics), but they are different languages, and Unity does not use CG. [*Writing Shaders*]. The game engine also has its asset store that contains a lot of free assets and 3d models. They can be used for the visual part of this work.

Unity also has a large online community, thousands of forum posts, and several subreddits, where people ask questions, and share their knowledge of the engine. All these factors make Unity a great choice for implementing this task. It is also important to note that both Unity3d and Unity refer to the same game engine.

2.3 Random number generation

Random numbers are an essential part of PCG. They are the key to generating the content, as they serve as the input to the rules of the generation. Modern computers are not capable of generating truly random numbers. An inefficient way to solve this is to use some external source like an atomic decay time in nature, as it is random [JamesE.Gentle, 2002]. Although this will work, it is incredibly inefficient. It is also possible to use physical processes on the computer to obtain random values.

This is where deterministic pseudo-random generators come useful. They can produce random numbers that have uniform distribution and appear to be completely random. Before starting, the generator needs to have an initial state. Defining it is done with a seed. A seed is a number or a set of numbers, depending on the type of generator. A common technique that most generators use is to take the current time of the device as its seed. In general, this technique works well. However, in some cases, when a function that uses a random number generator is called multiple times, the time values produced by each call of the function will be the same. In Unity3d a seed is an integer value. It serves as the point in the sequence where a particular run of pseudo-random values begins [*Unity Documentation*].To be able to generate different terrains, we can use different seeds, that will produce different results.

2.4 Noise in PCG

Noise is a powerful tool for PCG. It can be used to generate textures, simulate handwriting, clouds, and visual effects (VFX). It also is a great technique to use for terrain generation.

There are different types and algorithms that generate noise. Random value noise is the simplest one. It creates a grid (2d in this case), where each point has random float value between 0 and 1. The problem with using random value noise is that unlike nature, it has no consistency, meaning each point is independent of others. In order to achieve a more natural-looking terrain, the random grid must be smoothed out. This effect can be achieved using linear interpolation. The results, however, still usually look jagged.

Perlin noise is a type of noise that was developed in 1983 by Ken Perlin [Perlin, 1985]. At that time, he was working on an animated sci-fi movie "Tron". He later published his work in a SIGGRAPH paper. This type of noise is also sometimes called the gradient noise. It uses gradients and pseudo number generation to create a smoother and more realistic noise. In 2001 Ken Perlin made an improved implementation of the gradient noise and called it Simplex noise [*Noise Hardware*]. Simplex noise is more optimised, has lower computational complexity, can scale to higher dimensions, and it has no noticeable directional artifacts. In this work, I will try to use both the original Perlin noise, and the improved Simplex noise. Unity3d has Perlin noise implemented in its Mathf library. It only works in two dimensions, but this is not a problem for generating landscape.

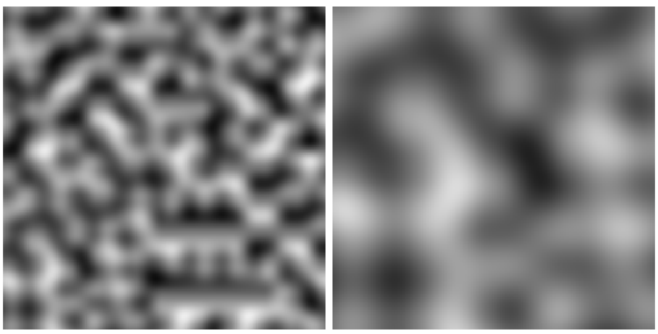
(A) Random value noise (B) Perlin noise (C) Simplex noise [Source]

FIGURE 2.1: Noise visualisations

The process of generating Perlin noise:

1. To generate a 2d noise with size width * height, we first need to create a grid with the same size. For this example, let us assume that width and height are equal, so the generated noise grid has a shape of a square. Each value in a grid has coordinates **y** and **y**. It is a common practice to use **u** and **v** instead of **x** and **y**, when working with noise or textures, so we will use this type of notation.

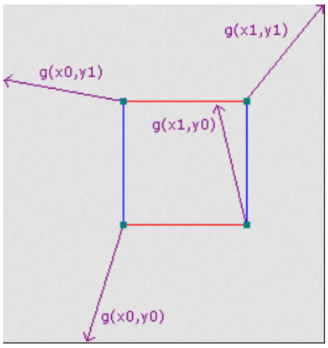
2. Now the grid can be divided into smaller subgrids to scale the noise. The division factor will be called the scale factor of the noise.



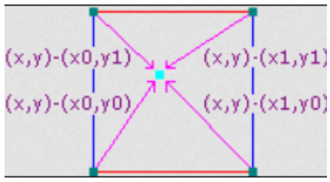
(A) scale = 20 (B) scale = 50

FIGURE 2.2: The impact of noise scaling

3. Use pseudo generation to generate gradient vectors, that face away from grid unit square.



(A) Random gradient vectors [Understanding Perlin Noise]



(B) Distance from cell to the point [Understanding Perlin Noise]

FIGURE 2.3: Gradient vectors

4. Calculate dot product between gradient vector, and the vector pointing to the point, at which noise value is calculated.

5. Do this for all four vertices. The result is four dot products **d1, d2, d3, d4**

6. Now to calculate the noise value at that pixel with coordinates **u** and **v**:

```
// lerp = linear interpolation
float x1 = lerp(d1, d2, u)
float x2 = lerp(d3, d4, u)
float noiseValue = lerp(x1, x2, v)
```

7. Linear interpolation is cheap, however, it is unnatural, and the results will look tiled. To make the newly generated noise to look better, we can apply an ease function. It will make the transition between tiles smoother and more natural. The preferred ease curve for this type of noise is $6t^5$ - $15t^4$ + $10t^3$, as defined by Ken Perlin. The curve is applied to the coordinates u and v.

Chapter 3

Implementation

3.1 Generating Noise

The first step is to generate noise that will be used to form the terrain. But before generating the noise, it is important to initialise the parameters that will control the rules of the generation.

- **Seed** ***(int)*** - a number that will initialise the state of the random number generator. For this work, we will set it to 1928371289.
- **Size** ***(int)*** - the size of the terrain that will be generated. Since for this work we set the width and height to be the same, the landscape will have a shape of a square.
- **Scale** ***(float)*** - a parameter that controls the scale of the generated noise.
- **Offset** ***(Vector2)*** - x and y offset of generated Perlin noise.
- **Height multiplier** ***(int)*** - the generated noise will always be in the range [0, 1]. In a mesh, a height difference between 0 and 1 will be hard to notice so that each noise value will be multiplied by this number.
- **Octaves** ***(int)*** - the number of octaves determines how many layers of noise we will generate. If we use just one layer of noise to generate terrain, it will not look realistic. One layered noise will look too smooth. To solve this problem, we need to stack multiple layers of noise together. Because Perlin noise uses a pseudo number generation, inputting the same values for multiple times will always return the same results. This means that each octave will be the same, and despite adding the noise, its final form will not change. This is where we have to introduce some random offset, that we will add to the sampling variables. However, picking a random offset every time we get the noise value at a point will not work either. Random offsets must be the same for every octave. An easy way to solve this problem is to generate random offsets before we start sampling the noise. This will ensure that every octave always uses the same offsets.

  Another problem is that simply adding the noise together is not enough. The next two parameters will help handle this.
- **Persistence** ***(float)*** - a number in the range [0, 1] that indicates how much each octave contributes to the final noise value. With every iteration, it adjusts the amplitude of the noise. At the start, the amplitude will be set equal to 1, and with every iteration, it will be multiplied by persistence value. Unless persistence value is set to 1, with each octave, the impact of the generated noise will

be decreasing, as amplitude decreases. For this work let us set the persistence value to 0.5.

- **Lacunarity *(float)*** - a number that measures how data fills space. This number determines how much detail each octave adds to the final noise value. This number adjusts the frequency of the noise. Just like with amplitude, frequency during each iteration it is multiplied by lacunarity. Setting lacunarity to 1.6 has shown the best results.

In order to comfortably pass the parameters between different components of the application, we will create a struct **TerrainParameters** that will store all the data. Now that we have the parameters that will define generation rules, we can start generating terrain. The first step is to generate the noise. The result is a 2d array where each value is in the range [0, 1].

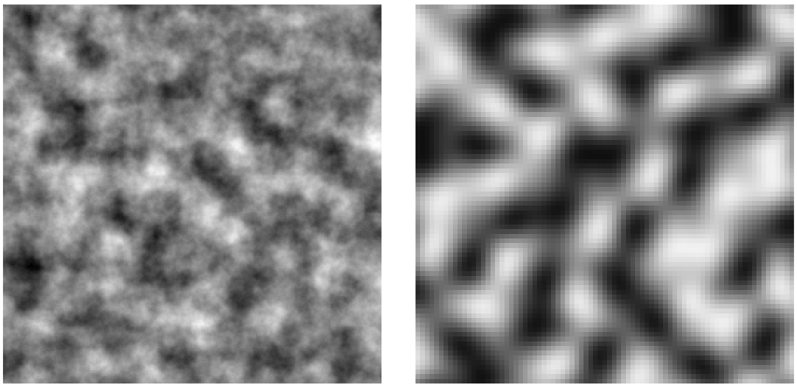
Noise generation code:

```
noise = new float[size, size];

for (var y = 0; y < size; y++) {
    for (var x = 0; x < size; x++) {
        float amplitude = 1;
        float frequency = 1;

        float noiseValue = 0;

        for (int i = 0; i < octaves; i++) {
            // adding randomly generated offsets is important
            float sampleX = x / scale * frequency + offsets[i].x;
            float sampleY = y / scale * frequency + offsets[i].y;

            float rawNoise = Mathf.PerlinNoise (sampleX, sampleY);

            noiseValue += rawNoise * amplitude;

            amplitude *= persistance;
            frequency *= lacunarity;
        }

        noise[x, y] = noiseValue;
    }
}
```

The value of noise returned by the ***Mathf.PerlinNoise*** function is modified during the process of adding extra layers of noise, so the values are not guaranteed to be in the range [0, 1]. To later use the noise for generation, all values must be within that range. This means that the noise has to be normalised. This can be done using ***Mathf.InverseLerp***. To do that highest and lowest noise values are needed. They can be obtained using C#'s LINQ ***Max*** and ***Min*** queries.

```
float[] castNoise = noise.Cast<float> ();
float max = castNoise.Max ();
float min = castNoise.Min ();

for (var y = 0; y < size; y++) {
    for (var x = 0; x < size; x++) {
        noise[x, y] = Mathf.InverseLerp (min, max, noise[x, y]);;
    }
}
```

Another way to generate noise, as mentioned above, is by using Simplex noise. The noise has an optimised implementation *Simplex noise demystified*. This approach was ported to C# in an open-source *Simplex Noise implementation*. To change the noise type, we must simply replace the call to **Mathf** library with the new noise function. However, the results turned out to be too sharp, even with extra layers of noise. So for this work, the original Perlin noise type will be used.



(A) Perlin Noise (B) Simplex noise

FIGURE 3.1: Generated noises

3.2 Generating a Mesh

Next step is to transform the 2d noise into a 3d mesh. The mesh must look like a ground in real landscapes. The generated noise already contains x and y values (the coordinates of the grid), so now we need to add another dimension for height. As we now work in 3d space, we will interpret the y coordinate of the noise as its z coordinate in 3d space. And the new y value will be calculated using the noise value at that point. To calculate this value, we must multiply the noise value by the height multiplier – a parameter that was defined earlier.

To create a mesh in Unity3d, we need to define its vertices, texture coordinates (also referred to as UV's), and the coordinates of the triangles that will form a mesh [*Creating a Quad*]

To display the mesh, we must create an empty GameObject in our scene, and add two components to it: MeshFilter, and MeshRenderer. As their names suggest, the mesh filter stores the mesh and its data, while the mesh renderer is responsible for drawing the mesh to the screen. Mesh renderer also allows to assign a material to the mesh, and control its shadows. We will leave the shadows parameters at default values. Now we need to define the mesh data.

- **Vertices** - The first step is to set up vertices that the terrain uses. All vertices will be stored in an array of type ***Vector3***. The number of vertices in a mesh is equal to ***size*** * ***size***. Each vertex will be in a form:

  Vector3 (x, noise[x, y] * heightMultiplier, y)

- **UV's** - Next, we have to define UV's of the mesh. In Unity, UV's are represented as an array of type ***Vector2***. UV's show how the mesh should be textured. This part is not very important, as we will not directly apply a texture to the terrain mesh.
  Using the same approach as with vertices, we need to divide each coordinate by the size of the grid. It is also important to cast the size to a float, so that the result doesn't get round up to integer value. We will store all UV's in an array. Its size is also ***size*** * ***size***. Every UV will be in a form:

  Vector2(x / float (size), y / float (size))

- **Triangles** - this step is the most important. We must define the vertices of the triangles that will make up the mesh. In Unity3d this data is stored in an integer array. In this array, each value is an index of a vertex that makes up the triangle. The number of triangles that a mesh with the size width * height can be calculated with the formula:

  ```
  int numberOfTriangles = (size - 1) * (size - 1) * 6
  ```

  The order in which points are described is also essential and must remain the same for every triangle so that the engine can properly rasterise the mesh. Vertices must be defined in a clockwise direction.

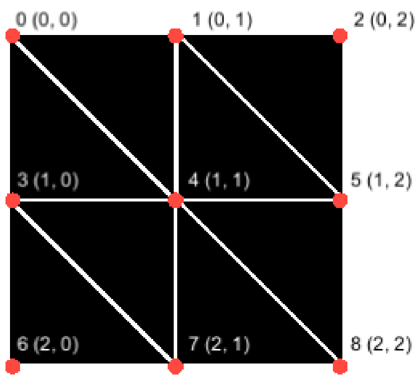


FIGURE 3.2: A mesh formed with triangles

Figure 3.2 shows a simple flat square mesh, that consists of eight triangles. The size of the mesh is **3 by 3**, and it has nine vertices. Let us define the first two triangles of this mesh. Their coordinates are:
t1 : {0, 4, 3} and **t2 : {4, 0, 1}**.
As you can see from the figure, the same pattern repeats for all other vertices except for the vertices on the right side of the mesh. We must include an if statement to handle this case. We will add vertices to the

triangles array only if the coordinates (marked in parentheses in the format (x coordinate, y coordinate) are smaller than **size - 1**. For every vertex that satisfies this condition, the triangles that it makes can be calculated with the formula:

```
// i is an index of the vertex in a mesh
t1 = {i, i + size + 1, i + size}
t2 = {i + size + 1, i, i + 1}
```

The process will be repeated for every vertex which x and y coordinate is less the size - 1, because vertices on the edge of the mesh do not have triangles to their right (as seen on figure).

After everything that is needed to create the mesh is ready, we can use Unity's mesh component, and pass the vertices, UV's, and triangles to it. After that, we recalculate the mesh's normals and tangents, so that the mesh is lit up and displayed properly. The created mesh object can now be assigned to the Mesh filter component in the scene. Now Unity will render it to the screen.

```
var mesh = new Mesh { vertices = Vertices, triangles = Triangles, uv = UVs
    };
mesh.RecalculateNormals ();
mesh.RecalculateTangents ();
```

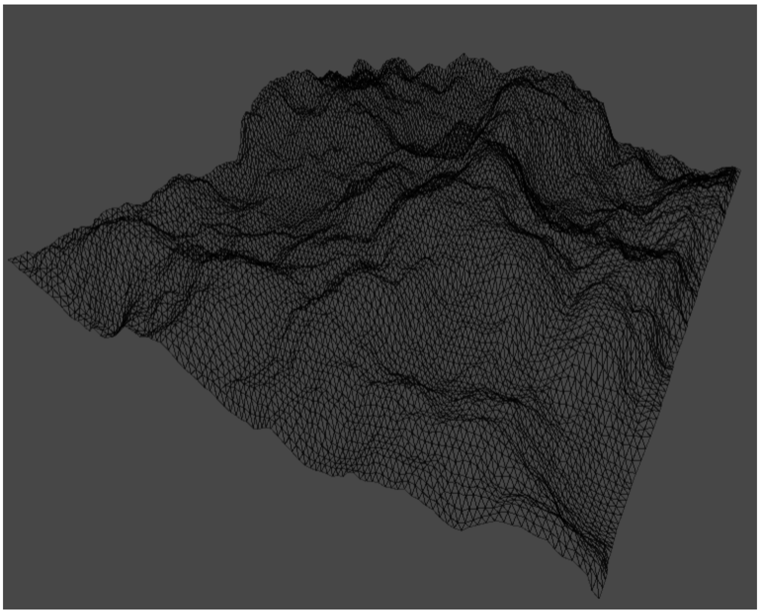


FIGURE 3.3: Wireframe view of the created mesh

3.3 Terrain shader

The newly generated mesh looks excellent. It has hills, valleys and plains, but it has no colours, and for the terrain to look realistic, it must have colours. We will achieve this by creating a custom surface shader. This type of shader uses "surface function" takes any UVs or data you need as input, and fills in output structure SurfaceOutput. SurfaceOutput describes properties of the surface (it's albedo colour, normal, emission, specularity, etc.) [*Surface shader manual*]. In our case, we only need the albedo, as we will use to set the texture. Albedo is a parameter that controls the base colour of the surface. To texture the mesh, we will use free textures from Unity's Asset Store or other copyright-free sources.

First we will divide the landscape into 5 separate layers:

1. Water
2. Gravel/Sand
3. Grass
4. Rocks
5. Snow

Now we assign each layer a height value and a texture that will be used to bring colours to the created terrain. Unity allows accessing the coordinates of a point that she shader is currently shading. We can use this to access the height of each point. To do this, we must include a ***float3*** variable named ***worldPos*** in the shader's Input struct. Now the engine will automatically fill this variable.

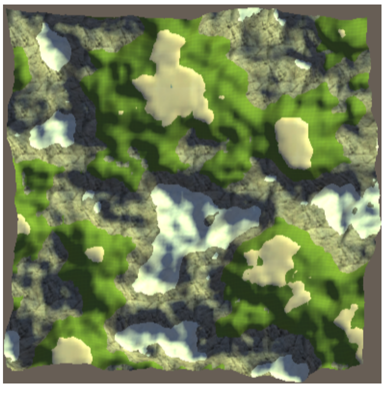
```
struct Input {
...
// Now to access the height we can use worldPos.y
float3 worldPos;
}
```

We are going to have several textures on one mesh. For this to look good, we need to have some blending between all the textures. The blending effect will add smoother transitions between heights and their textures. We can control the intensity of blending with a float parameter. To properly blend the textures, we will use the approach suggested on the Unity Forum. It uses a custom struct called blendingData. It will store the data about the height that we are currently sampling, and the resulting data that we will later pass into the albedo. First, we will need to get the data of the texture that we have assigned earlier. This can be done with the function **UNITY_SAMPLE_TEX2D**. As an input, it takes in the texture, and its UV float2 values. After all the textures are initialised, we have to blend them, using the height parameters that we have defined. Each texture is treated as a separate layer. For each layer, we assign a new height value to our struct. Next, we calculate the colour value using the blend intensity parameter, and then we use linear interpolation of the calculated result with the currently stored result, and the result value of the layer that we are currently blending. This process has to be repeated for every texture. This approach allows us to add or reduce layers.

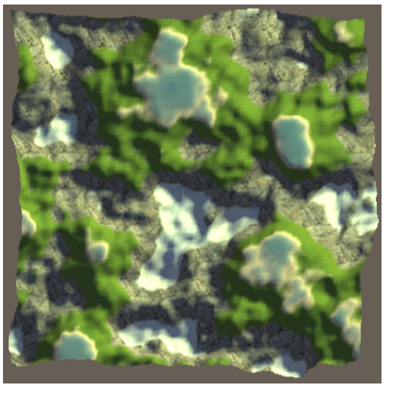
To make sure that our mesh looks nice even when its size gets big, we will add texture tiling. Unity's materials have support for this feature. Instead of projecting an entire texture on the mesh, it will tile in on both axes. The number of tiles is determined by user input. Having it set to **10** for both axes gives good results. But since we have a lot of water around the edges, let us set the tiling number to 30 for the water texture, so it does not look too stretched.

Blending function:

```
blendingData BlendLayer(float4 layer, float layerHeight, blendingData bd)
{
    bd.height = max(0, bd.height - layerHeight);
    float t = min(1, bd.height * _BlendSharpness);
    bd.result = lerp(bd.result, layer, t);

    return bd;
}
```



(A) Blending disabled



(B) Blending enabled

FIGURE 3.4: The impact of texture blending

To make Unity use our newly created shader, we must create a new Material [Unity Manual, Materials], and then in the shader dropdown menu, choose the shader that we have just created. Next step is to apply this material to the MeshRenderer component, and now Unity will use our shader to render this mesh.

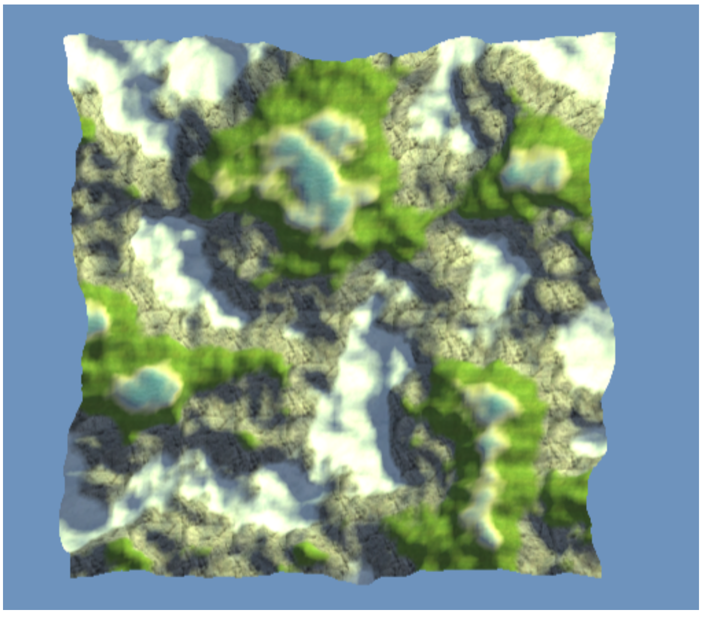


FIGURE 3.5: Aerial view of the textured mesh

3.4 Improving the realism of the mesh

Terrain now has colours and looks realistic. However, its edges end with sliced mountains. Right now, the landscape consists of just one tile. To fix this, we can generate more tiles on each side of the terrain. Having many tiles stacked side by side will make the landscape look massive. This widespread implementation allows developers to create infinite worlds, as we can add more and more tiles for players to walk on. However, it is a very computationally heavy task that requires a lot of optimisation. Having so many tiles would mean that the GPU would have to do a lot of extra work. 3d games with large maps use a technique called "level of detail" (LOD) switching [Computer Graphics, ISY, LiTH]. It allows reducing the number of triangles of the meshes that are far away from the player so that the GPU load is smaller. LOD switching can be done dynamically [GPU based dynamic geometry LOD]. But for this work, let us choose a different approach - turn this landscape into an island using another level of artificial noise. This technique is similar to falloff map - a tool used in game development, computer graphics and textures. The most common use case for the falloff map is reflections.

For this work, we will generate a new layer of noise, and use it to turn the generated terrain tile into an island. To achieve this, we can subtract the values of the extra layer of noise from the values of the generated Perlin noise and re-generate a mesh. The noise must significantly decrease the values on the edges of the created terrain, resembling an island.

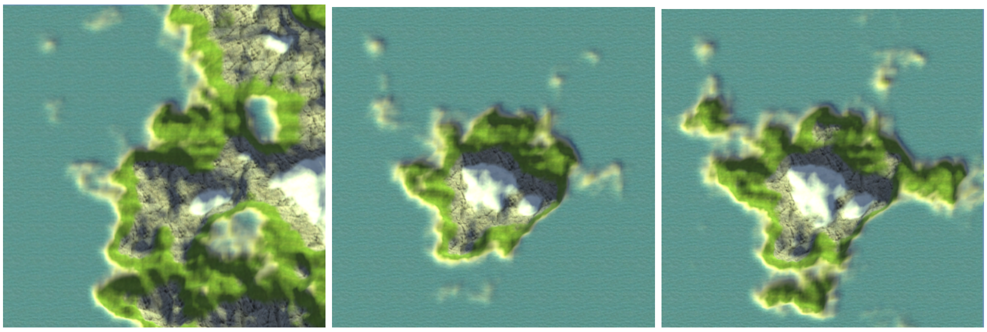
I have picked three types of noise to modify the mesh.

(A) Distance to the right edge (B) Distance to the closest edge (C) Distance to the centre

FIGURE 3.6: Generated noise.

The first falloff map [Figure 3.3 A] can be used to create an effect of an edge. It will transform the noise to have low values (darker) at the right side, and higher values on the left side (brighter). Visually the terrain will be textured as water on its left side, making the terrain look like a real edge of the continent. However, this type of noise only generates high values on one of the sides.

The next two noises are much better for creating islands, as they both have high values close to all four edges. The square shape of the second noise [Figure 3.6 (B)] does not look as realistic as the round shaped noise of the third type [Figure 3.6 (C)]. Also, it is not as smooth as the third noise type. The third noise type allows the terrain to have higher values closer to the edges and fills up more space with the land. For this work let us choose the circular noise to serve as an extra layer.



(A) (B) (C)

FIGURE 3.7: Terrain with the new layer applied.

3.5 Forest Generation

To generate a realistically looking forest, placing trees at random is not enough. There must be consistency, and trees can't overlap. Also, the y coordinate that represents the height of a tree in a 3d scene must match the y coordinate of the previously generated mesh, so that trees do not float in the air, or grow underground.
Cellular automata – a discrete abstract computational system [Cellular Automata, Stanford Encyclopedia of Philosophy], also famous for Conway's Game of Life [Conway's Game of Life FAQ] can be an excellent way to solve this. This system (let us interpret it as 2d grid, where each cell has coordinates x and y) consists of cells that can have two states: active or inactive. By defining rules, we can change the state of the system. To change the state, we must iterate through every cell in the grid, and check how many neighbours it has (let us call the number of neighbours n). If n is smaller than some threshold integer value (t), then we turn off the cell. Otherwise, when n is larger then the threshold value the cell gets turned on. This process can be repeated multiple times to achieve the best results.

To change the state of the system, we must define some initial state, because if all points have equal states, then no changes will occur. For this work, we can randomly set the states of each cell. Let us introduce another variable p in the range [0, 1] that will represent the probability that a cell's state will be active when the system starts.

After the initial state of each cell has been set, we can start to change the state of the system. The goal is to turn the system into a value noise grid that will define where a tree can be placed. We will interpret active cells as cells that have trees and inactive cells as cells that do not have trees or other objects inside. The process of changing the state of the grid can be repeated multiple times to achieve better results. A variable k represents the number of iterations that changed the state of the system.

```
int[, ] grid = new int[size, size];

// set the initial state
for (var y = 0; y < size; y++) {
    for (var x = 0; x < size; x++) {
        float chance = Random.value;
        if (chance < p) {
```

```
            grid[x, y] = 0;
        } else {
            grid[x, y] = 1;
        }
    }
}

// change the state k times
for (var i = 0; i < k; k++) {
    for (var y = 0; y < size; y++) {
        for (var x = 0; x < size; x++) {
            int n = GetNeighbours (x, y);
            if (n > t) {
                grid[x, y] = 1;
            } else {
                grid[x, y] = 0;
            }
        }
    }
}
```

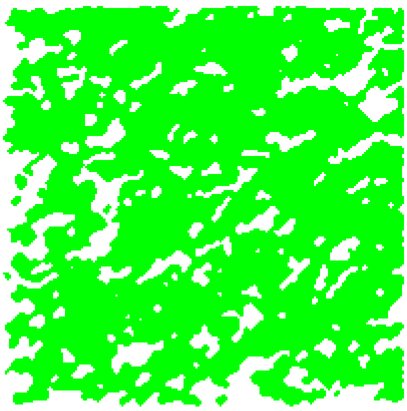


FIGURE 3.8: Cellular automata

The results [Figure 3.8] look beautiful and realistic. Each green point is a tree, and each white point is an empty space. The trees are connected, and there are forest openings, which make it look realistic. However, there is a problem with this approach. Every tree is represented by a cell in a generated grid, rather than a coordinate. Being bound to a cell means that the distance between every tree in the grid will always be the same, and that is not how trees in real life forests grow. Of course, each tree's position can be altered by some random offset, but as with the case of random noise for terrain, the results do not look realistic, and it can potentially create a lot of problems.

Another way to place trees is by using Poisson Disk sampling. This algorithm allows placing points on a grid, with any dimensionality procedurally. It also ensures that all points are at least r units away from all other points, meaning that no 2 points can overlap, and unlike in the previous implementation with Cellular Automata, the positions are not bound to a fixed grid.
In this work, we will use Fast Poisson Disk Sampling in Arbitrary Dimensions [Robert Bridson, University of British Columbia]. It is an optimised implementation of this

algorithm, that is guaranteed to take ***O(n)*** time to generate **N** Poisson disk samples.

For this task, we need to introduce new parameters:

- **Radius *r* (*float*)** - the minimal distance between each point.
- **The number of dimensions *n* (*int*)** - in this case, it is 2, as we need to place trees on a 2d grid. The height value of each point is determined by the terrain noise generated earlier.
- **The number of attempts *k* (*int*)** - this number determines how many times an algorithm will attempt to generate a new neighbour for a given point before discarding it.

Generation process:

1. Calculate cell size ***c***, using the formula

```
float c = r / sqrt(n) = r / sqrt(2)
```

2. Initialize a 2d grid using the cell size and the size of the terrain

```
float gridSize = Mathf.CeilToInt(size / c);
int[,] grid = new int[gridSize, gridSize];
```

3. Create two lists: one to keep track of created points, and another one to keep track of active points. A point is active if it can still have neighbours.

4. Place the first point at the centre of the grid and add it to the active list. The coordinates for this point can be calculated by dividing the grid size by 2.

```
Vector2 centerPoint = new Vector2 (size, size) / 2;
```

5. While the active list is not empty, for each iteration pick a random point **i** from active points list

   (a) For **k** times attempt to spawn a random point. Pick a random angle to calculate a random direction:

   ```
   float angle = random value * Mathf.PI * 2
   Vector2 direction = Vector(sin(angle), cos(angle))
   ```

   Now initiate a point at the position:

   ```
   Vector2 newPoint = i + direction * Random.value (radius, radius *
       2)
   ```

   This means that the new point will be from **r** to **2r** units away from the currently active point.

   (b) Check if a point can exist at those coordinates. If it overlaps with other points at that cell in the grid, discard this point and attempt to spawn a new point. Otherwise, spawn a point, and add it to the list.

   (c) If a point was spawned, break the loop, and pick a new random point.

(d) If after k tries no point was spawned, this means that the point **i** cannot have new neighbours, so we must remove it from the active list, and pick a new point **i**, if any are available.

(e) If no points are available, then the generation is completed.

We now have a list of points where a tree can be placed. Now we need to add height to all these points. The height of each tree is obtained using the same Perlin noise that was used to generate the terrain.

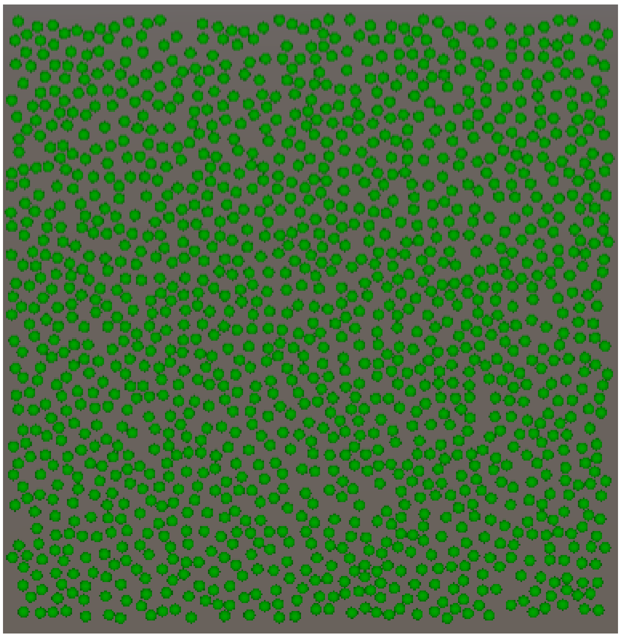


FIGURE 3.9: Poisson disk sampling

3.6 Instantiating the trees

Now that we know where a tree can be placed let us instantiate a 3d mesh of a tree at that point. For the visual part of this work, I will use the free low polygon tree models from the Unity3d asset store [LowPoly Trees and Rocks] [Low-Poly Simple Nature Pack]. Both packages contain textures, materials, and 3d models of different types of trees and other natural objects. To optimise the performance of the game, let us spawn a preset collection of trees at each point, instead of generating a single tree. The rotation on the Y-axis of each collection will be randomised. This change will ensure that the forest looks more like real-life forests. The rotation will be a random float number in the range [0, 360]. Another improvement is to create multiple collections, that will be picked randomly during the generation process. As we use the same seed for initialising the state of the randomiser that places trees, the generated forest will always look the same, as long as the seed remains unchanged.

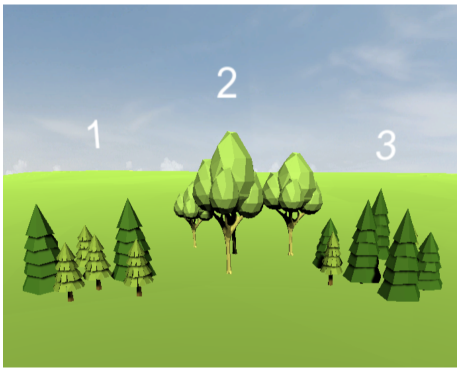
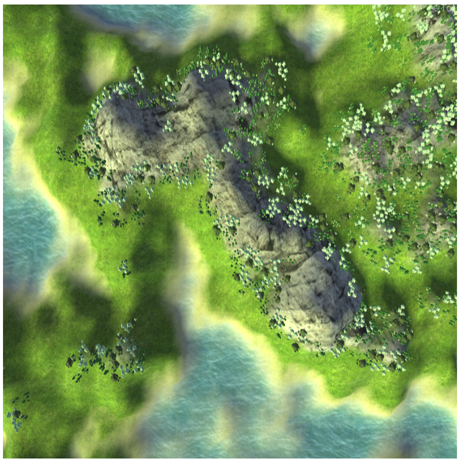


FIGURE 3.10: Examples of the collections

For the next step, we will need to introduce two new variables: **lowerBound** and **upperBound**. They will represent the lowest and the highest points at which a tree can be instantiated. Without them, the entire terrain (including the lower parts that are textured as water) will be covered with trees and other objects. Since we do not want that, a tree will only be spawned when its Y-axis coordinate falls within the given range. Through experimenting, I have found that setting the lower bound to 0.37 and the upper bound to 0.8 produces the best results. The water and the rocky, snowy mountains do not have any trees growing now.



(A) Aerial view

(B) Forest zoomed in

FIGURE 3.11: View of the generated forest

Chapter 4

Combining the Generations

To combine the two parts of this work (mesh and forest generation), we will need to create a controller that will call the generation functions of each component. The controller will also store the generated noise, as both components use it. Now we will create controllers for the Mesh Generator and the Forest Generator. They will both have a Generate method that accepts noise and other parameters as an input. The function that calls the Generate method for other components can be set to execute at the start of the application, or assigned to a button. For debugging purposes, we can also add a boolean variable to turn one of the components on or off. This is useful for choosing the best generation parameters.

TerrainGenerator's fields

1. MeshRenderer
2. MeshFilter
3. TerrainParameters (serialized field that we can set from the editor).
4. bool generateMesh - used to turn the generation on or off

ForestGenerator's fields

1. float lowerBound
2. float lowerBound
3. GameObject[] treeCollections - an array of preset tree collections
4. bool generateForest - used to turn the generation on or off

Chapter 5

Results

Now that everything is finished and the components work well together, we can experiment with the parameters of the generation, to see what kind of results will the generation produce. Here are the examples of the landscapes generated with approaches and techniques suggested in this work.

(A) seed = 2 (B) seed = 16 (C) seed = 128

FIGURE 5.1: Final results produced by different seeds

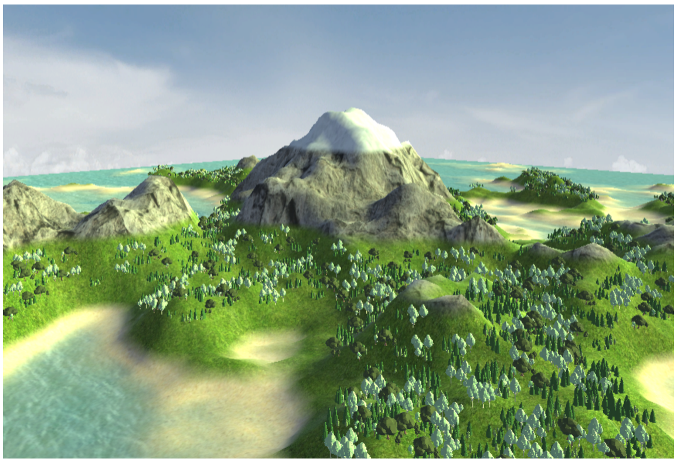


FIGURE 5.2: A beautiful mountain at the center of an island

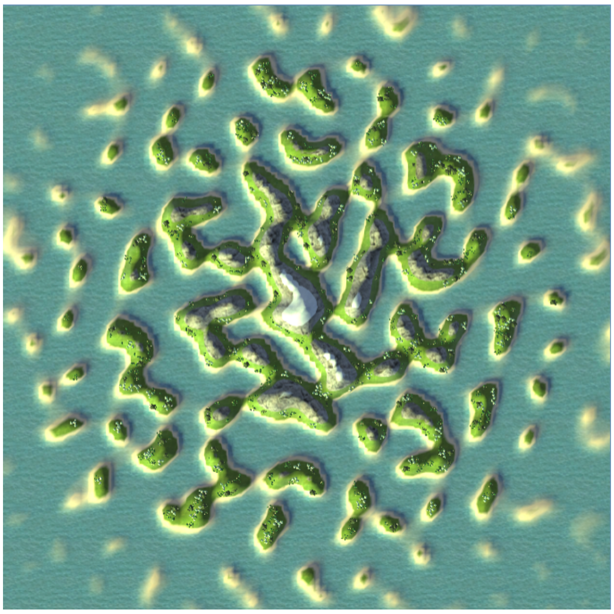


FIGURE 5.3: An unsuccessful attempt to use simplex noise

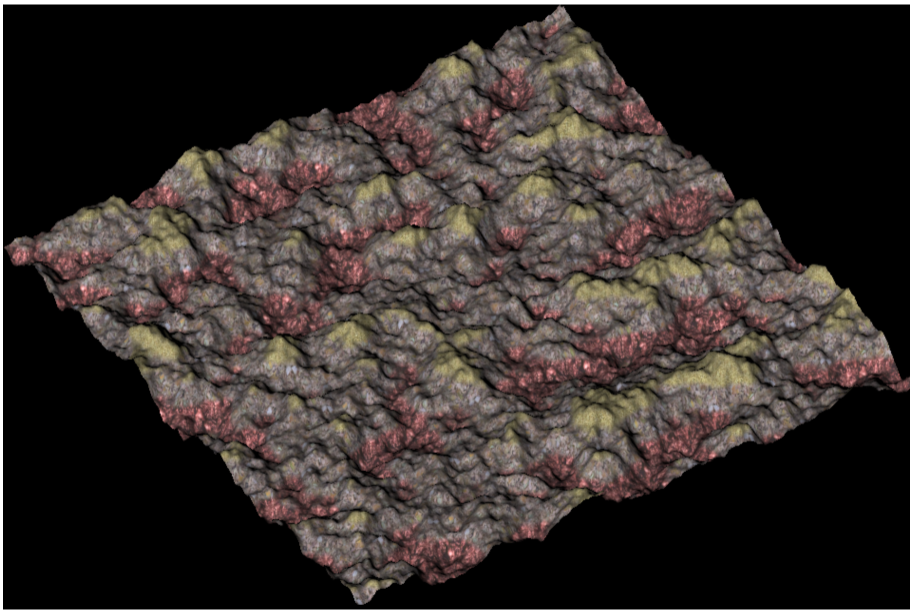


FIGURE 5.4: Experimenting with textures, to make the landscape look like an alien planet. Textures from Yughues Free Nature Materials

Chapter 6

Conclusion

The goal of this work was to generate a realistically looking 3d landscape that consists of mountains, lakes, plains and is populated with different trees, that make up forests. To achieve this, we split the work into two parts: generating and texturing terrain and then generating a forest. Both parts required us to generate noise or a layout and then transform it into a 3d scene. For the terrain we used Perlin gradient noise, that was later transformed into a 3d mesh. Next step was applying a material with a custom surface shader that brought colours to the terrain. Having textures allowed the ground to look realistic. The forest was generated by procedurally placing trees using Poisson disk sampling. The algorithm ensures that no two points overlap. Later the points were projected onto the terrain mesh, and at each coordinate, trees were placed. A falloff map was used to turn the terrain into an island. It subtracted the noise values at the edges of the grid. The result is a realistically looking island that has lakes, grass, sand, mountains with snow on top, and a lot of different trees that form forests.

This work shows a few ways of using procedural generation algorithms and tools for video games. It shows that using PCG for creating in-game content can be very useful and can generate creative results. PCG can also save a lot of time for developers, artists, and game designers, and saved time usually means saved money. Now the generated landscape can be used for different types of video games. It could be modified to create an infinite world for a survival game like Minecraft, or we can stick with single or multiple islands, for a survival game like Stranded Deep. Another great application is using this landscape for a flight simulator game.

Procedural content generation is truly a fantastic tool for game development. Although the primary goal of this work was achieved, it can be further expanded with additional features like an extra layer of noise to indicate the weather (similar to biomes in Minecraft). We can use fractals to generate the plans and trees, instead of using preset 3d models. The possibilities are endless, and this is what is excellent about PCG.

Bibliography

Perlin, Ken (1985). "An Image Synthesizer . Courant Institute of Mathematical Sciences New York University". In: 19.3, pp. 1–10. DOI: https://dl.acm.org/doi/pdf/10.1145/325334.325247.

JamesE.Gentle (2002). "RANDOM NUMBER GENERATION AND MONTE CARLO METHODS". In: George Mason University. Chap. 1.1.

Gortler, Steven J. (2012a). "Foundations of 3D Computer Graphics". In: MIT Press. Chap. 22.2.

— (2012b). "Foundations of 3D Computer Graphics". In: MIT Press. Chap. 1.1.

DAHLSKOG, STEVE (2016). "PATTERNS AND PROCEDURAL CONTENT GENERATION IN DIGITAL GAMES". In: Malmo Univercity. Chap. 2.2.

Julian Togelius Mark J. Nelson, Noor Shaker (2016). "Procedural Content Generation in Games". In: Springer. Chap. 1.9.

Benjamin, Ward. *Simplex Noise implementation*. URL: https://github.com/WardBenjamin/SimplexNoise.

Gustavson, Stefan. *Simplex noise demystified*. URL: http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf.

Newzoo. *2018 Global Games Market Report*. URL: https://web.archive.org/web/20191102122939/https://newzoo.com/wp-content/uploads/2016/03/Newzoo_2018_Global_Games_Market_per_Device_Segment.png.

— *Top 25 Public Companies by Game Revenues*. URL: https://newzoo.com/insights/rankings/top-25-companies-game-revenues.

Perlin, Ken. *Noise Hardware*. URL: https://www.csee.umbc.edu/~olano/s2002c36/ch02.pdf.

Talbot, Carrie. *2011's Minecraft becomes the second best-selling game of August 2019*. URL: https://www.pcgamesn.com/minecraft/sales.

Unity. *Creating a Quad*. URL: https://docs.unity3d.com/Manual/Example-CreatingaBillboardPlane.html.

— *Surface shader manual*. URL: https://docs.unity3d.com/Manual/SL-SurfaceShaders.html.

— *Unity Documentation*. URL: https://docs.unity3d.com/ScriptReference/Random.InitState.html.

— *Unity User Manual*. URL: https://docs.unity3d.com/Manual/index.html.

— *Writing Shaders*. URL: https://docs.unity3d.com/Manual/ShadersOverview.html.