A winter terrain - Procedural images, TNM084

Wilma Axelsson, wilax550

January 2022

Contents

1 Introduction **2**

2 Method **3**
2.1 Terrain . . . 3
2.1.1 Perlin Noise . . . 5
2.1.2 Fractal Brownian Motion . . . 5
2.2 Snowfall . . . 5
2.3 Snow accumulation . . . 6
2.3.1 The Normal Texture . . . 6
2.3.2 The Dot Product calculation . . . 7
2.3.3 The Sample Texture . . . 8
2.3.4 The Smoothness Map . . . 8
2.3.5 The Snow Color . . . 9
2.3.6 Noise Calculation . . . 9
2.3.7 Displacement Map . . . 10

3 Results **11**

4 Discussion **14**
4.1 Results . . . 14
4.2 Further developments . . . 14

5 Conclusion **16**

Chapter 1

Introduction

One major aim in computer graphics is to create as realistic scenes as possible for the user to look upon. This also means that the scenes should be able to vary to consist of anything between a desert, a forest, an ocean, or a winter terrain. In this project, the last named has been created, namely the winter terrain. The terrain has been created with procedural generation, which is a method that speeds the process of creating complex objects, such as a varying terrain. The procedural generation needs to have three important properties: speed, randomness, and controllability. The method of procedural generation also uses procedural noise such as Perlin noise, to generate randomness in the terrain. In this project, the procedurally generated winter terrain has been implemented using Perlin noise .

Chapter 2

Method

The project was implemented in Unity, using C# as well as Unitys' Shader graphs. For rendering the scene with the shaders, *Universal Render Pipeline* (URP) was used.

The three main goals of the project were to create a procedural terrain using Perlin noise, a particle system of snowfall and snow accumulation on the terrain based on the snowfall. To create a more controllable scene for the user, controls for changing various properties of the scene was also implemented. These controls include changing the terrain, activating a snowstorm, and changing the amount of snowfall.

2.1 Terrain

The base of the terrain was created using the default 3D object *Terrain* in Unity, which gave a flat terrain or plane at first. The Terrain object consists of various data, named *terrainData*, which are properties such as the height, depth and width of the terrain heightmap. Using these properties, the terrain heightmap could be changed using various functions.

The terrain heightmap was altered using a script which included functions such as *RandomOffset()*, *GenerateTerrain()*, *GenerateHeights()* and *CalculateHeight()*. The RandomOffset() function generates random values in specific ranges for the three variables scale, offsetX and offsetY, see figure 2.1. This function was required for a random generation of the terrain each time the scene was built.

The GenerateTerrain() function generates, as its name suggests, the new terrain based on set values height, width and depth. The height variable of the terrain was then changed through the GenerateHeights() function, where a float array was used to store the new heights, see figure 2.2.

```
    void randomOffset() //Generate a new, random terrain each time
    {
        offsetX = Random.Range(0f, 9999f);
        offsetY = Random.Range(0f, 9999f);

        scale = Random.Range(5f, 40f);
}
```

Figure 2.1: The function RandomOffset() which generates random values for the terrain

```
TerrainData GenerateTerrain(TerrainData terrainData)
{
    terrainData.heightmapResolution = width + 1;

    terrainData.size = new Vector3(width, depth, height); //x, y, z

    terrainData.SetHeights(0, 0, GenerateHeights());

    return terrainData;
}

float[,] GenerateHeights()
{
    float[,] heights = new float[width, height];
    for(int x = 0; x < width; x++)
    {
        for (int y = 0; y < height; y++)
        {
            heights[x, y] = CalculateHeight(x, y); // CalculateHeights returns the Perlin noise for new Height value
        }
    }
    return heights;
}
```

Figure 2.2: The functions GenerateTerrain() and GenerateHeights().

The new height points for the terrain heightmap were calculated through the CalculateHeight() function. Here, the x and y coordinates are transformed, based on the height, width, scale and offset values of the terrain, and passed into the *Mathf.PerlinNoise()* function [3], see figure 2.3. Also, we use the variables *amplitude*, *frequency* and *octaves* to create *Fractal Brownian Motion* (FBM) [1].

```
float CalculateHeight(int x, int y) // Using Perlin noise
{
    float xCoord = (float)x / width * scale + offsetX;
    float yCoord = (float)y / height * scale + offsetY;

    int octaves = 3;

    float amplitude = 0.5f;
    float frequency = 1f;
    float noise = 0;

    for (var oc = 0; oc < octaves; oc++)
    {
        noise += amplitude * Mathf.PerlinNoise(xCoord * frequency, yCoord * frequency);
        amplitude *= 0.5f;
        frequency *= 2f;
    }

    return noise;
}
```

Figure 2.3: The function CalculateHeight(), containing the noise calculation for the new terrain coordinates.

2.1.1 Perlin Noise

The Mathf.PerlinNoise() function, as seen in figure 2.3, is based on creating Perlin noise, which is a type of gradient noise that creates a pseudo-random pattern. It is a function where we return a float, i.e., the Perlin noise, based on the width and height (x and y coordinates) that are sent in to the Perlin noise function. This float is therefore based on a 2D plane, and it varies between the values -1 and 1. Perlin noise was chosen as it is a smoother noise with random qualities, and therefore it creates a more smooth but still random pattern over the object, which helps in creating a terrain with hills and valleys. [2]

2.1.2 Fractal Brownian Motion

We also use FBM, or *fractal noise*, to create more finer detail and more varying heights in the scene, than by just using Perlin noise. In FBM we use a number of octaves (e.g. three octaves), which determines how many iterations of noise we apply, and we decrease the amplitude by half and double the frequency each iteration, as seen in the code in figure 2.3. If we would increase the number of octaves, we create finer detail but also more self-similarity in the terrain. Too much self-similarity in the terrain would give a less varying and natural terrain, and so we keep the number of octaves at a reasonable low number.

2.2 Snowfall

The snowfall was implemented using a particle system in the Unity program. The particles/snowflakes were initialized with lifetimes, to not make the program crash with too many particles in the scene. The particle system also consisted of a *Noise* variable, which included properties such as *Frequency* and *Strength* that controlled the intensity of the noise. Through increasing these

properties, the particle system would spawn particles with more noise and the snowfall would look more 'chaotic', resulting in a snowstorm effect. Because of this, these properties were increased if the user would press the button "Enable Snowstorm".

Other than Strength and Frequency, the *Octaves* property also existed. This property would have helped in scaling the noise, but it did also degrade the performance of the program more, and so this property was kept at a low value (i.e. value 1).

2.3 Snow accumulation

The layer of snow on the terrain was implemented using a Unity Shader graph. In this shader graph we make use of both a vertex shader and a fragment shader for the terrain. The fragment shader goes through a noise calculation to create a white, slightly dirty color of the snow. The vertex shader is used to create a layer of snow on top of the terrain, which consists of creating displacement using noise, so that the terrain rises in height as more snow falls. There are in total seven factors or calculations that contribute to the shader: The *Normal Texture*, a *Dot Product calculation*, the *Snow Color*, the *Snow Buildup (the Noise Calculation)*, the *Sample Texture*, the *Smoothness Map*, and the *Displacement Map*.

2.3.1 The Normal Texture

The Normal Texture uses a *Base Normal* texture as well as *Tiling* and *Offset* values to calculate the normals for the fragment shader, see figure 2.4.

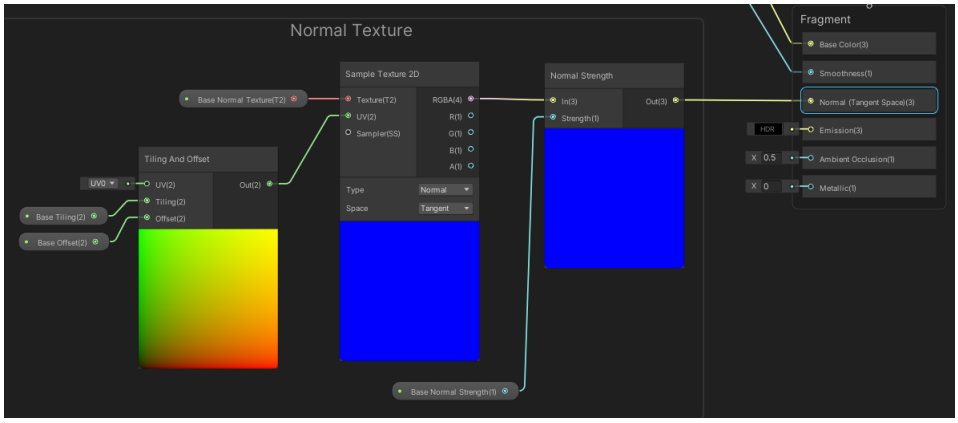

Figure 2.4: The Normal Texture.

2.3.2 The Dot Product calculation

The Dot Product calculation is necessary for the direction of the snow, i.e. how the snow will land on the objects it hits. It takes use of a Snow Direction variable, which is set to a 3D vector with x equal to 0, y equal to 1 and z equal to 0 in order to have the snow placed in positive y direction (upwards), see figure 2.5.

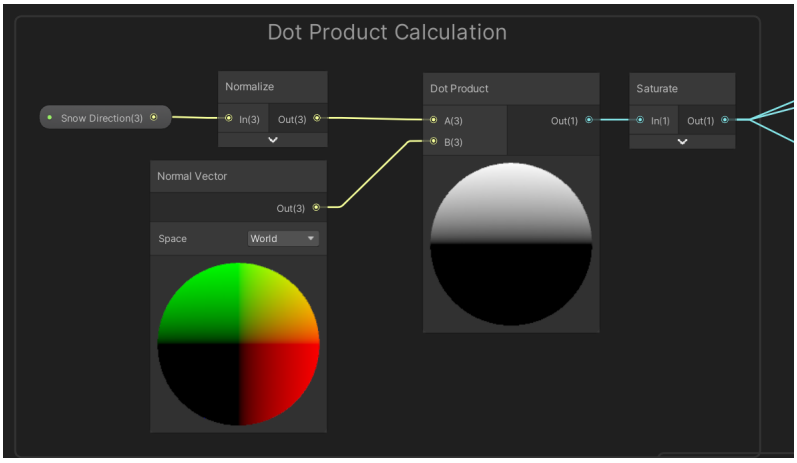

Figure 2.5: The Dot Product calculation

2.3.3 The Sample Texture

In order to create a more realistic snow, the *Sample Texture* uses the Noise Calculation result as well as the *Snow Color* to create the snows' color. This is done by using a *Lerp* node, which linearly interpolates between the Sample texture and the Snow color by the Noise calculation value. The result from the Lerp node is then connected to the Base Color socket of the fragment shader, meaning it gives the snow its final color. See figure 2.6 for the whole process.

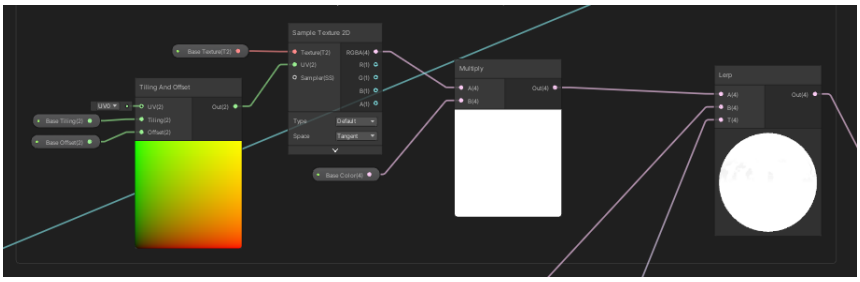

Figure 2.6: The Sample Texture.

2.3.4 The Smoothness Map

In order to give the snow buildup a more smooth, snowy color, a smoothness map was implemented, see figure 2.7. Here, we also use the Lerp node to interpolate between the Noise calculation value and the Smoothness texture, to connect its result with the fragment shaders' Smoothness socket.

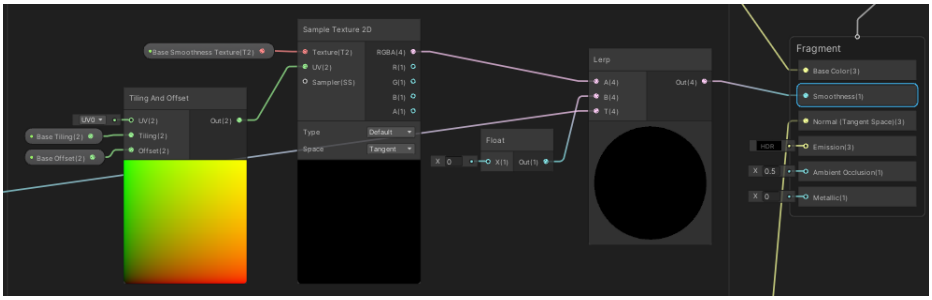

Figure 2.7: The Smoothness Map.

2.3.5 The Snow Color

In the Snow Color process, we want to simulate a snow color that varies depending on the amount of snow layered in a location. For example, if the amount of snow has had little time to build up, the snow should have a faint, barely visible white color. We therefore use a *Remap* node to vary the *Snow Color* (which is a variable of type *Color* that is naturally white) depending on the *Snow Color Noise Size* and *Snow Color Noise Strength*. See figure 2.8 for the whole process.

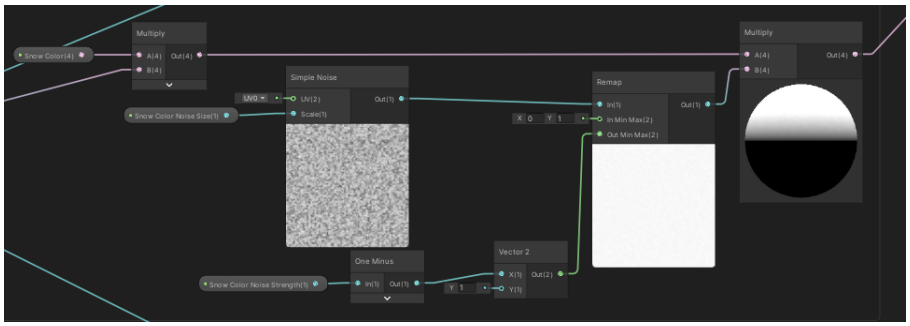

Figure 2.8: The Snow Color.

2.3.6 Noise Calculation

The Noise Calculation is shown in figure 2.9, which for instance consists of two *Simple Noise* nodes that is mixed with a *Smoothstep* node to create a noisy effect. The Smoothstep node receives the variables *Snow Amount* and *Snow Blend Distance*, which determine how much of the snow is colored onto the terrain. The result of the Smoothstep node is then multiplied with the Dot Product calculation in order to build the snow in the correct direction, and the result is sent into a *Saturate* node to fix the final values between 0 and 1. The result from the Saturate node can then finally be used for the calculation of Snow Color and Smoothness Map.

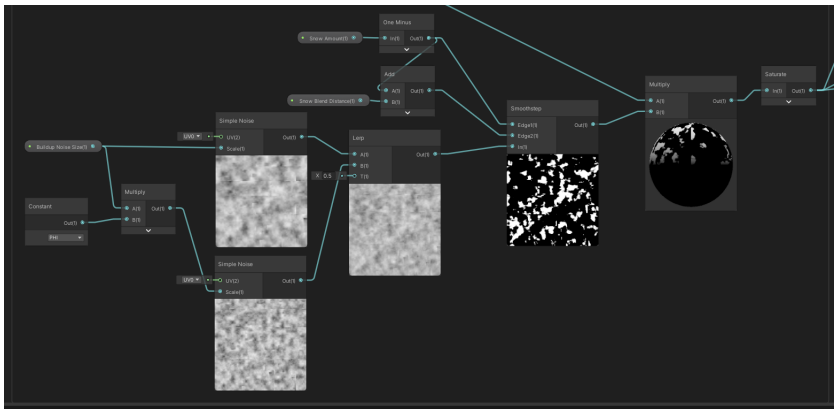

Figure 2.9: The noise calculation for both vertex and fragment shader

2.3.7 Displacement Map

A displacement map was created in the Shader graph for creating a higher terrain as the snow accumulates. Here, we use a *Position* node that defines the terrains' position, and we connect it to the same Noise Calculation process as shown in figure 2.9. We multiply the result with a *Normal Vector* node, and connect this with the Position node to finally connect the result to the *Vertex Position* socket in the vertex shader. See figure 2.10 for a full view of the Shader graph.

Displacement maps are useful as it uses a simple idea to alter the geometry of the terrain or object, by changing each point along the normal of the terrains' smooth surface into new positions. The normals stay roughly the same but the surface is different, which makes the concept of displacement maps similar to bump maps [2].

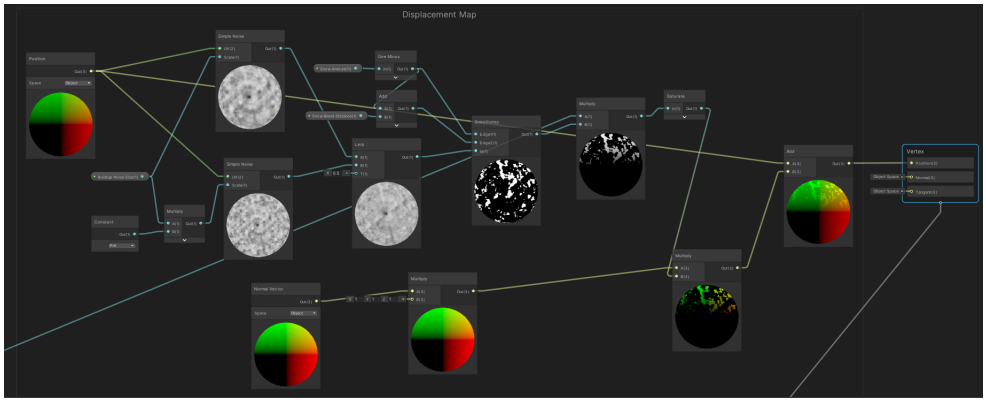

Figure 2.10: The displacement map created to alter the terrains' positions.

Chapter 3

Results

The results show a procedural terrain, with snowfall and snow accumulation, which were created mainly through the use of Perlin noise. A scene with only the procedurally generated terrain is seen in figure 3.1. By changing the slider "Snow Amount", we can increase or decrease the snow amount to fasten or slow down the snow accumulation.

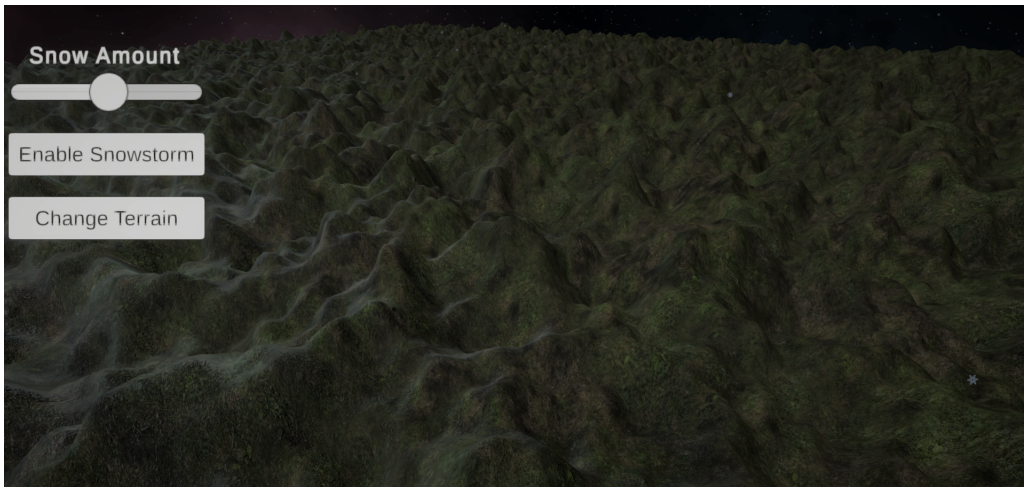

Figure 3.1: A procedurally generated terrain (without snow).

A terrain with snow and the snow accumulation process is shown in figure 3.2, where you can see the snow building up the terrain. The snowstorm effect can be seen in figure 3.3.

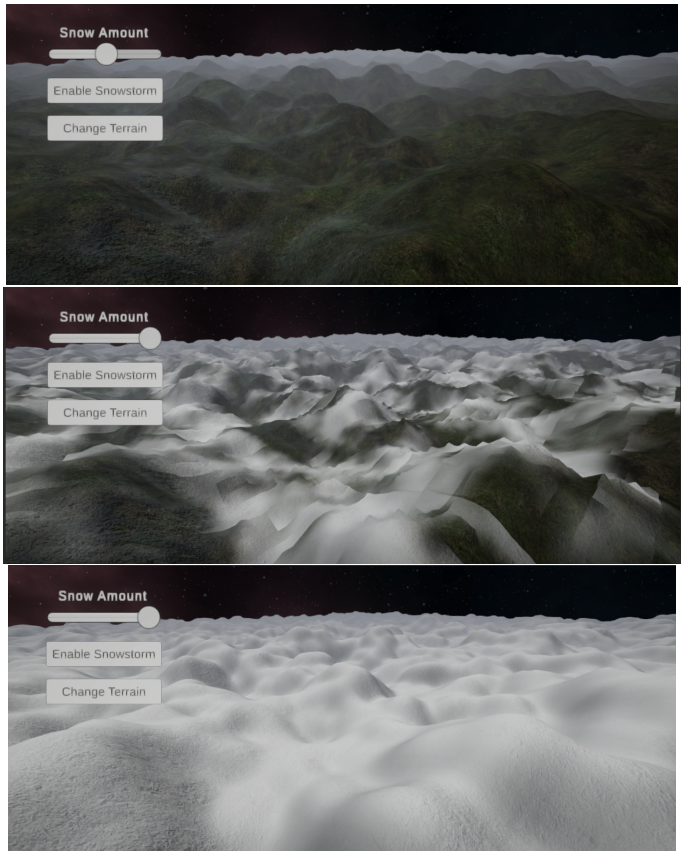

Figure 3.2: A procedurally generated terrain (with snow). The top image shows terrain with no snow, the middle image shows some time after as more snow has landed, and the bottom image shows the terrain after the snow has fully landed.

Figure 3.3: The scene after enabling snowstorm.

Chapter 4

Discussion

4.1 Results

The results in figure 3.1 show a terrain which has a large variety of mountain tops, but it does not have hills that "stand out" more, i.e. the hills maximum height are all quite similar. This is because we had to set a maximum height to the terrains' height, and the fractal noise could create new heights only up to that maximum. In order to create even higher mountain tops, the maximum height of the terrain would have to be set higher.

In figure 3.3 we see that the noise used created a good effect of snowstorm, as the snowflakes increased in size as well as numbers. As the snowfall covers a large area, spawning more snowflakes and adding more noise to their movements makes the program slightly slower, and so the number of particles needed to be lowered slightly (maximum of 2000 particles).

4.2 Further developments

There are still some aspects of the project that could be further tested or improved. This includes the implementation of the vertex shader for the snow layered on the terrain, as it only gives a smaller rise in terrain. This is because the vertex shader had a limitation in changing the terrains' positions, since too much changing gave a visually unnatural growth of snow. The snow accumulation also gave slight square patterns in the terrains' texture, as can be seen on the terrain in the middle image in figure 3.2, which made the terrain seem slightly more unnatural. To fix this, a smoothness factor could perhaps have been added to vertex shader.

Also, we can see some aliasing problems in the texture of the terrain, see figure 4.1. Here, we see that the further away we see the terrain, the snows' texture receives diamond square patterns. Two solution of this problem would be to for example use a bump map, which would create a bumpy surface and the

aliasing would not be seen, or use *mipmapping* to create progressively smaller textures of the original texture. [2]

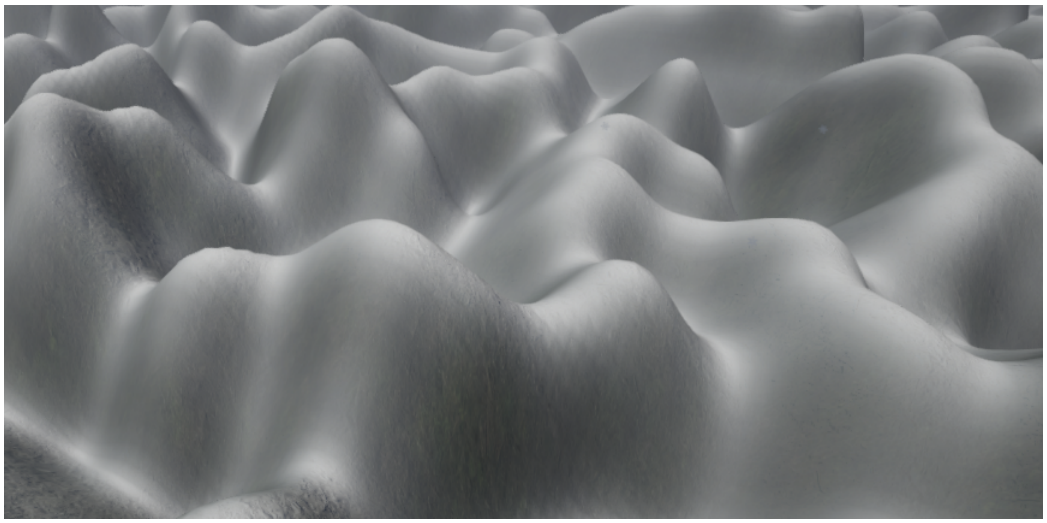

Figure 4.1: Aliasing can be seen in the snows' texture (diamond square pattern).

Chapter 5

Conclusion

The resulting project, shown in figures 3.1, 3.2 and 3.3, show that the use of Perlin noise and Unitys' Shader graphs and particle systems can very well create simple terrains and snow accumulation. The use of displacement map for the terrains' vertex shader also worked well and it was easy to implement.

As a final conclusion, the project was able to create the procedural terrain in varying sizes with a low rendering time. The snow on the terrain was easily altered using the Shader graph and the snows' properties could be controlled by the user in order to create a more dynamic environment. This shows that procedural generation is a very useful tool in computer graphics, especially when creating large terrains and snow.

Bibliography

[1] Jen Lowe Patricio Gonzalez Vivo. Fractal brownian motion. https://thebookofshaders.com/13/, 2015. Online: hämtad 05-januari-2022.

[2] Peter Shirley Steve Marschner. *Fundamentals of Computer Graphics*. Taylor Francis Group, LLC, 2016.

[3] Unity. Mathf.perlinnoise. https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html, 2021. Online: hämtad 04-januari-2022.