Graphical Rendering of Dynamic Weather Systems

Lyle Madette Dotingco
180112607

7th May 2021

BSc Computer Science

Supervisor: Dr Gary Ushaw

Word Count: 11,720

Abstract

Whilst real-time rendering of virtual weather conditions has been thoroughly investigated before, modern development platforms are constantly improving in efficiency and computational costs. This means that techniques that were once too resource-intensive can now be employed for a fraction of the demand.

In this paper, I propose different ways particle systems, shaders and scripts can be used within Unity to produce realistic weather efficiently. The motivation for this project stems from this issue being a central concern across all features of a game.

A summary of the project contents is documented below, with the decisions on design and development justified. The results and analysis are recorded later in the document, along with the evaluation and the potential future expansions to the research of this project.

Declaration

"I declare that this dissertation represents my own work except where otherwise stated."

Acknowledgements

I’d like to thank my personal tutor and supervisor, Dr Gary Ushaw, for his support throughout this project and university life. Also, Dr Graham Morgan and Dr Rich Davison for their invaluable advice.

To my friends for the company and the memories made throughout this journey. I thank you dearly.

Finally, to my parents; thank you for the countless sacrifices you’ve made. The patience, the belief, the guidance and the love that allowed me to come this far. This one’s for you.

Table of Contents

Table of Figures

1 Introduction

This section aims to introduce the project to the reader and provide them with the details of the work that was done and the motivation behind the project. On top of this, the high-level aim will be broken down into manageable objectives to easily assess the success of the project as a whole.

1.1 Purpose

A certain degree of realism is required to fully immerse the player within the world they're playing in. Weather is a fundamental aspect in game design because it enhances immersion by a tenfold due to pathetic fallacy, ensuring that the player's emotions are directly influenced by natural phenomena. As gaming is a form of escapism, the weather being an unplanned feature of our daily lives also brings in that unpredictable element and can further impose problems for the player to deal with as part of the game progression.

A great example of this is The Legend of Zelda: Breath of the Wild, where if the main character sets fire to dry grass caused by the sun, it creates an updraft that they can ride on and use as a form of transportation. This level of attention to detail shows the untapped potential development companies have when implementing gameplay mechanics. With Breath of the Wild winning Game of the Year in 2017, this clearly shows that there can only be advantages towards adding weather effects to games. [1]

1.2 Motivation

One of the earliest recorded evidence of weather being implemented into games was Utopia (1981) on the Intellivision video game console [2]. The premise of Utopia is something most real-time strategy (RTS) players are familiar with; two players take control of one of the game's two islands and spend gold bars in constructing buildings, boats, farms and housing for their people. Being an RTS game, different conditions mean that the player generates income based on the randomly generated rain which passes over the player's farms. Rain results in crops being watered, but hurricanes and tropical storms could destroy player progression too. [3] This meant that weather was not only an aesthetic decision but also an integral part of gameplay.

From the time of writing this paper, Utopia was released nearly 40 years ago. Figure 1 above shows the graphics used to portray rain clouds.

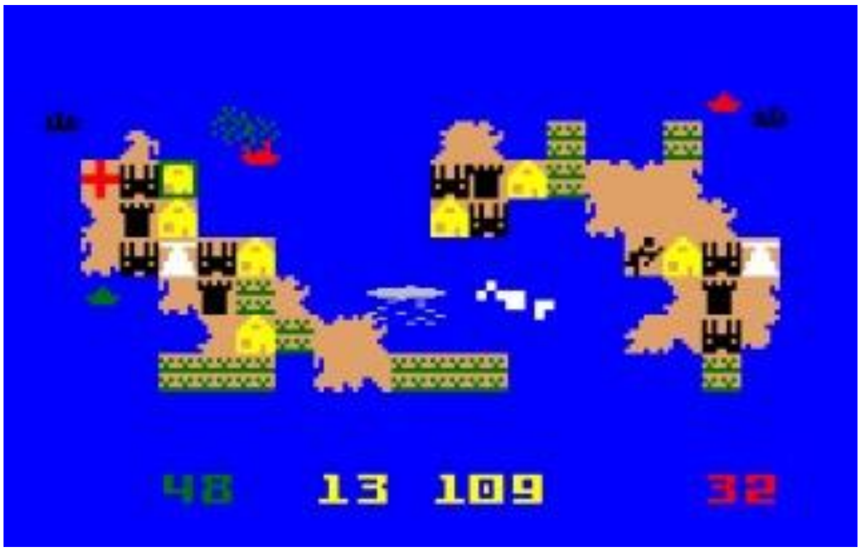


Figure 1 Rain clouds shown in Utopia gameplay [47]

Games developed on the Intellivision were written in assembly language, whereas in 2021 game engines most commonly use C# or C++. Whilst the difference between assembly language and high-level languages are negligible in terms of application, nowadays the processing power and memory of computers are significantly more powerful compared to 4 decades ago. As the years go on, there's no sign of stopping.

A study carried out by Hans Moravec titled "When will computer hardware match the human brain?" [4] investigates the prospect of computers matching the intellectual performance of humans, and the time scale estimated to reach that point.

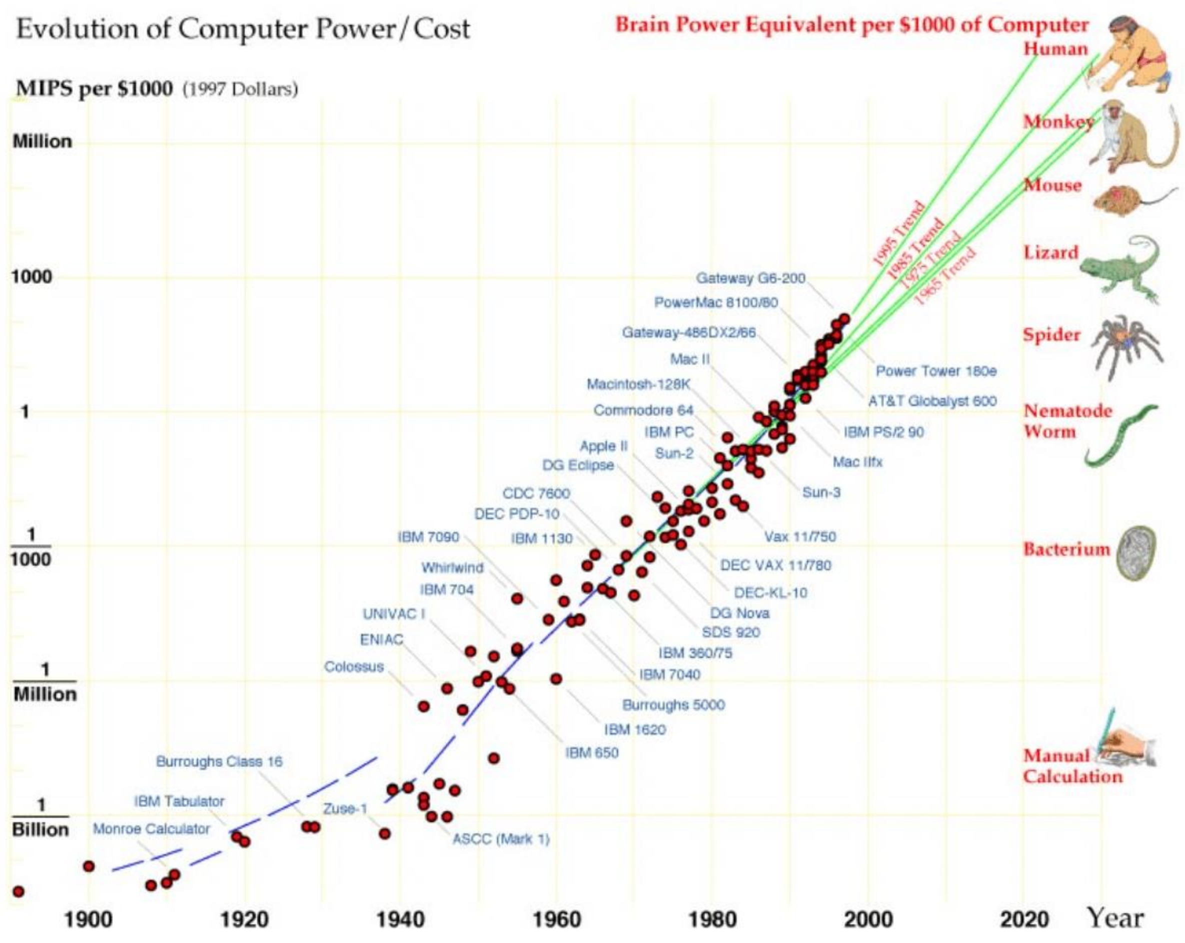


Figure 2 Moravec's graph on the "faster than exponential growth in computing power"

Figure 2 shows the evolution of computing power over cost, and despite being published in 1998, it was estimated that by 2020 a $1000 computer will at least have the processing power of the human brain. Moravec explored the developments of computers between 1900 to 1940 where the speed of calculation saw a "thousand-fold" increase. Yet, between 1940 and 1980 saw a further "millionfold" increase.

While spending $1000 on a PC may seem costly, game engines would not need a computer that powerful to run. Therefore, this further adds to the rationale behind this project. Computers, software and game engines are always improving in processing costs and runtime efficiency, resulting in the constant need for improvement in rendering techniques.

1.3 Project Aim

Investigating the production of realistic weather systems efficiently–exploring the balance between performance and appearance.

1.4 List of Objectives

1.4.1 Research

Engage in the background reading on rendering weather to form an initial understanding of how to reproduce effects on a standard setup.

Investigate the current market and how atmospheric games use weather systems effectively, and the relationship between performance and appearance.

1.4.2 Identify

Research and identify key weather systems to implement, choosing carefully to produce a well-rounded project that encompasses the main weather-based immersion methods.

1.4.3 Design

Design an appropriate game world to apply the systems to. This should be a custom environment with peaks and dips to simulate a natural environment. This would allow the opportunity to generate a snow cover or show the boundaries between a lower, foggy area and an elevated and clear one.

1.4.4 Implementation: Snow

Implement standard snowfall and an additional snowstorm/blizzard condition. This includes the snow particle system and a more intense and denser counterpart. Render clouds to set a start point for the particle system, and add a procedural snow cover over the terrain.

1.4.5 Implementation: Rain

Implement a rain shower and an additional rainstorm over the game world. This includes the rendering of rain particles for a standard shower, and then a more intense rainfall. Clouds that gradually turn darker to signify a rainstorm, occasional sounds of thunder crashes, and changes to the skybox. Finally, rain ripples and a glossy appearance over the terrain.

1.4.6 Implementation: Fog

Implement a heterogenous fog blanket over the game world that encompasses the terrain, varying in density. This should reduce the visibility of the player-controlled camera.

1.4.7 Testing

Repeat testing of the program, ensuring each weather system works independently according to these objective aims, and also influence each other if they're to be activated in succession.

1.4.8 Evaluation of Performance

In the final stage of development, I will evaluate if the project runs smoothly at 60 frames-per-second (FPS) and in full screen. A clear written assessment will be created to note down the performance, and compare the project against these objectives–calculating the success of my project.

1.5 Approach

I'm focusing on 3 distinct weather conditions: snow, rain, fog because I feel they are the most distinct states. These also have sub-conditions to further increase complexity, such as a snowstorm which will include opportunities to add sound to increase immersion.

Through this, I will be documenting my findings on how to reduce memory usage, and decrease computational costs enough to still reach that degree of realism that weather systems provide.

1.6 Schedule

The project schedule for the first and second semester with module deadlines and details of the key objectives previously outlined is visualised below. Figure 3 expands on the time frames of each activity.

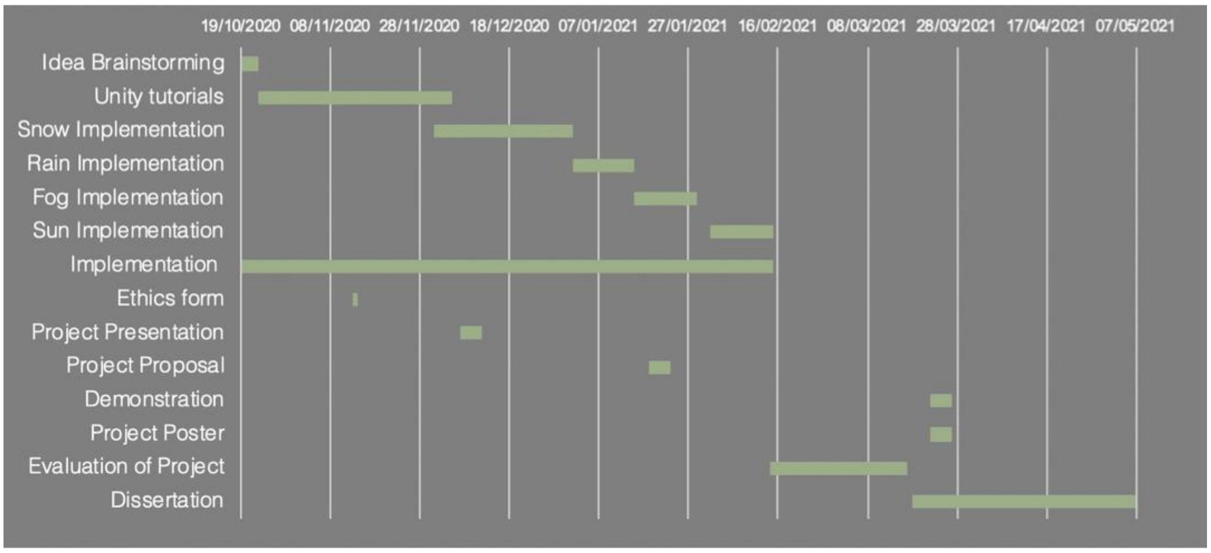
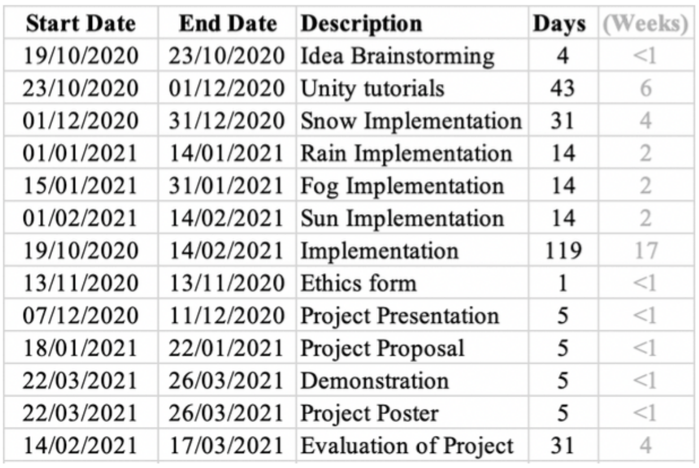


Figure 3 Project schedule



Start Date	End Date	Description	Days	(Weeks)
19/10/2020	23/10/2020	Idea Brainstorming	4	<1
23/10/2020	01/12/2020	Unity tutorials	43	6
01/12/2020	31/12/2020	Snow Implementation	31	4
01/01/2021	14/01/2021	Rain Implementation	14	2
15/01/2021	31/01/2021	Fog Implementation	14	2
01/02/2021	14/02/2021	Sun Implementation	14	2
19/10/2020	14/02/2021	Implementation	119	17
13/11/2020	13/11/2020	Ethics form	1	<1
07/12/2020	11/12/2020	Project Presentation	5	<1
18/01/2021	22/01/2021	Project Proposal	5	<1
22/03/2021	26/03/2021	Demonstration	5	<1
22/03/2021	26/03/2021	Project Poster	5	<1
14/02/2021	17/03/2021	Evaluation of Project	31	4

Figure 4 Detailing duration for each item on the project schedule

1.7 Report Structure

This project will be broken down into various sections.

Section 1, this section: Introduction, is to introduce the purpose, aim and objectives of the report.

Section 2, Background Research, containing various studies and sources that allowed this project to commence.

Section 3, Implementation, covers the development plan, the technology utilised within the project, and the implementation itself. This includes the reasoning behind design decisions and in-depth details on project progression.

Section 4, Results, contains the project outcome alongside performance tests. These tests will be visualised in graphs and tables for easier viewing.

Section 5, Evaluation, discusses whether the project has been successful in terms of meeting the aforementioned aims and objectives.

Section 6, Conclusion, is the final analysis of the dissertation and the techniques I implemented, what has been learnt and what could have been improved on in hindsight. It will also cover further developments on the project to advance it further.

2 Background Research

This section explains the research carried out for this project, with the fundamental concepts explained that relate to it.

2.1 Academic Works

A review of existing academic papers that relate to my topic and provide essential understanding towards rendering weather effects accurately.

2.1.1 Realistic Real-Time Rendering [5]

This paper explores how to portray rain droplets and streaks in line with real-life physics. With the researchers unsatisfied with the relation of the prevalence of rain, and how inaccurately represented it is within games.

This provided me with initial insight into the depth of where my project could go, with Rousseau et al examining the physical properties of raindrops and their refraction. They also briefly investigated how snow and fog can be translated into a game setting, which helped me build an initial understanding of how my project could render these conditions within Unity's capabilities.

2.1.2 A Spectral-Particle Hybrid Method for Rendering Falling Snow [6]

This paper describes the issues with the aim of an accurate depiction of snow and the huge computational costs that come with it. They propose a spectral-particle method which is similar to a billboard approach, where you replace distant objects with images. In doing so they were able to show a denser snowfall with minimal cost, and in the end, the result showed a more accurate depiction than increasing particle count itself.

Because I plan to also implement the storm counterpart to snow and rain, this would require me to increase the density of these conditions. Understandably, increasing the particle count would directly influence the power needed to run the program. Therefore, I needed to find ways to reproduce that effect efficiently, and this paper provided me with an alternative.

2.1.3 Computer Modelling of Fallen Snow [7]

This paper investigates how to model a thick layer of snowfall on the ground. The accumulation pattern differs upon the surfaces it lands on, and this introduces "flake flutter," the notion that parts of an object that has no direct exposure to the sky can also be coated in snow.

Producing falling snow with particle systems makes up only half of the challenge. There is also the procedural accumulation of snow upon the game world. At first, it seemed like a straightforward task, but upon reading this paper I realise that many factors such as the directions of the object matter when rendering a blanket over their material. The introduction of flake flutter presents another consideration for the project.

2.1.4 Foggy Scene Rendering Based on Transmission Map Estimation [8]

This paper explores the realistic rendering of fog in game development, taking extra caution to avoid producing homogeneous fog with a uniform appearance throughout the blanket. Introduces Perlin noise and its role in generating heterogeneous fog with varying degrees of shape and density.

This helps to provide an understanding of how to produce fog accurately. The paper identifies different factors to consider upon development, and I've learnt to steer away from solely using particle systems that would produce homogeneous results. It is especially helpful considering that the user will be able to move the camera around the world space, and the fog should cater to that movement by changing its appearance.

2.1.5 Real-Time Rendering of Volumetric Clouds [9]

This dissertation paper investigates alternative methods for rendering clouds other than having a library of cloud images. This would be a sufficient solution but the cost of having a large library with high-resolution images would be excessive. Like with the previous paper, this also considers Perlin noise but with the combination of an inverted Worley noise to introduce cloud-like shapes.

Within the thesis are visual goals that the author has outlined, which inspired my implementation of clouds. This includes varying size and coverage, consideration of lighting, and the importance of soft shadows.

2.1.6 Rendering of 3D Dynamic Virtual Environments [10]

This paper describes a custom framework developed to encompass the main features of a dynamic 3D virtual environment. This includes detailed insight into the terrain, skyboxes, and sound. Moreover, this considers past research on reproducing 3D environments.

The research conducted consolidates information from previous investigations on this topic, which is why I chose this reading since it is similar to the position I'm currently at with the project. The method in which they generate the environment is of importance, utilising heightmaps and textures. But overall, their framework contains methods on the dynamic simulation of weather, clouds, rain and lightning too, which I'll take great inspiration from in terms of the direction they took, and apply it to my understanding of Unity.

2.2 Algorithms

Though most of this project will be developed using pre-defined game objects in Unity that I can calibrate such as particle systems, a portion of the systems will be made possible through C# scripts. In this sub-section, I will explain how I plan to use the following algorithms.

2.2.1 Snow Building Script

While procuring pre-textured assets that are rendered with already existing snow is possible, for game purposes it may not be the most helpful. AAA games developed by major publishing companies would most likely have a blank game world free from environmental effects to modify the location directly through gameplay. Therefore, it is important to understand how to build these effects anew—accurately and efficiently.

This script will be using the object's normal value. Normal, in the terms of 3D graphics geometry, refers to the perpendicular vector value to a point on the rendered surface. [11] This is used by the computer to generate the appropriate lighting effects as it determines the direction each point is facing. To continue, the normal value will be used to calculate which parts of the object will have their texture modified to have "snow" placed on it, based on the direction that the surface faces.

There will be two normals: the world plane normal, and the snow shader normal which defines where the snow should be. The script aims to gradually shift between these values.

The snow normal is a separate value so that the user can programmatically alter the source of the snow particles to consider wind direction. Therefore, portraying the weather effects accurately through the snow building on the terrain at an angle, if the user wishes for the snow to fall this way.

2.2.2 FPS Counter

To calculate the frames per second of the system, I will utilise the Update function that is built into each Unity script. This function is called after every frame so long as MonoBehaviour is enabled. (MonoBehaviour is a base class that must be used to create scripts through C#. [12]) As Update is called per frame, I can use Unity's built-function: "Time.unscaledTime" which returns the time in seconds since the start of the game, but since it's within Update, it will return the time taken to complete the last frame.

2.2.3 Memory (RAM) Counter

Similar to the FPS counter, I will also be tracking the memory usage of each system for comparison later on in the project. To do this, Unity's Profiler class includes a function called "GetTotalReservedMemoryLong" which returns the memory reserved by Unity. This will be used to show the memory being used by the program during runtime and is integral to calculating the performance between systems.

Again, this function will be called after every frame like FPS. This means that I can get an accurate reading of how much memory is being used.

3 Implementation

Implementation is the primary part of this project. I will go into detail on the processes regarding bringing the project to life, the reasoning behind my decisions, and the techniques I considered but ended up abandoning.

3.1 Planning

Because of the nature of this project, I attended fortnightly meetings with my dissertation supervisors to discuss objectives for the next two weeks. An agile approach to the project – implementation, documentation, and then presenting for feedback.

Figure 5 below shows the agile model adapted for my project.

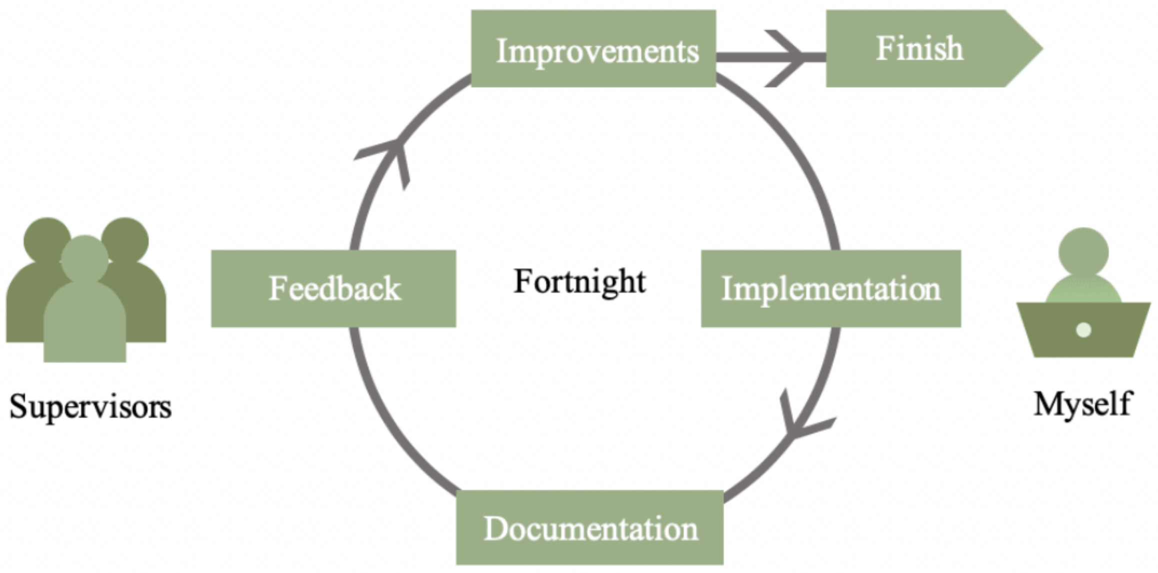


Figure 5 Project approach

I was advised to begin the implementation of the systems as the majority of the time spent on the project will be in development. From Figure 3 picturing my proposed project schedule, I followed my deadlines and the project was a success on the whole.

3.2 Technology

3.2.1 Development machine

Naturally, I would have used the University's computers which are tuned to handle large computational tasks. However, because of the COVID-19 pandemic, I had limited to no access to equipment on campus. This meant that development was hindered due to technological restrictions.

Remote desktop access to these computers was possible but very inconvenient considering the scope of the project. Therefore, all aspects of this project were developed on macOS.

Computers in the Urban Sciences Building (Home of the School of Computing):

- Core i5 6600 3.30GHz Quad-Core
- 16GB Memory
- NVIDIA GeForce GTX 1050 2GB
- Windows 10 Enterprise 2016, 64-bit
- Western Digital Blue 500GB

My personal laptop (MacBook Pro 13-inch, 2017):

- Core i5 3.1GHz Dual-Core
- 8GB Memory
- Intel Iris Plus Graphics 650 1.5GB
- macOS Big Sur
- SSD 500GB

This clearly shows that my personal computer is weaker in processing power with only a dual-core compared to a quad-core i5. On top of this, I have half of the RAM capacity, and the graphics card difference is great.

Using the GPU UserBenchmark [13] online, I compared the power between the two graphics cards and these were some of the significant results (GTX 1050 vs Iris Plus 650):

GTX 1050 has:

- 193% better lighting effects.
- 205% better reflection handling.
- 217% faster multi rendering.
- 178% better peak lighting effects.
- 174% better texture detail.

These confirm that rendering was impacted.

It is well known that MacBook's are hardly the custom when it comes to game development as their inner components typically aren't modular, and the cards built into these laptops are a fraction of the power found in custom work/gaming PCs.
Despite this, there were no major difficulties developing on the laptop. The largest disadvantage was that my machine couldn't handle large rendering requests, which I will expand on later in this dissertation.

3.2.2 Unity

Founded in 2004, Unity Technologies created a new game engine: Unity, supporting designing, buying and importing digital assets to use on the program. It is an industry-standard development platform, used not only for 2D and 3D game development but also simulations and interactive experiences [14]. It utilises C++ at runtime, and C# when scripting.

"…world's most powerful real-time development platform." [15]

It seemed the natural choice when choosing the platform to base the project on because of its popularity within game companies. The Unity CEO has also claimed that half of all the games ever built have been built on Unity [16]. I also have experience in using Unity from previous University degree modules and so my familiarity was also a defining factor in the choice.

Instead of using a dedicated benchmarking tool, Unity supports scripting so instead, I created a script that calculated FPS during runtime—this was briefly mentioned in Section 2.2.2, and will also be expanded on later in the document.

3.2.2.1 Pipelines

Projects in Unity has to conform to one of the two rendering pipelines available: High Definition Render Pipeline (HDRP), or Universal Render Pipeline (URP). The issue between these is that they are not compatible with each other. Features found in one pipeline cannot be used inside another, making the choice between them an important one.

Officially released in the 2019.3 patch, HDRP is the more modern version of the two, aiming to target modern compute shader compatible platforms [17]. This allows the user to use dynamic global illumination, ambient occlusion, and volumetric fog.

URP was formerly known as Lightweight Render Pipeline (LWRP). The main advantage with URP is that it is older and therefore the most stable version of the two, and the most widely adopted just in terms of longevity. Although URP misses a lot of the more favourable features outlined above, many of these elements can be hardcoded with scripts.

I chose to begin the project in URP when playing around with the platform and then attempted to use HDRP to make use of the Physically Based Rendering (PBR) Graphs.
I will expand on these choices later on in Section 3.3: Development.

3.2.2.2 Shaders

Materials and shaders go hand in hand. To properly light and colour each game object, a shader must be applied to it. These are small scripts that contain the information the GPU needs to calculate the colour of each pixel [18].
Unity provides standard shaders with options that can be modified to change a material’s properties.

However, sometimes the desired output is outside of the capabilities of the base shaders. This would require you to write a shader script in an external shading language such as Cg (a high-level shading language developed by NVIDIA [19]). This is where HDRP shines. Because of the pipeline, Unity now supports Shader Graphs which is a more accessible way of creating custom properties for your materials, and its visual interface means it requires minimal coding which cuts down on the additional learning curve that learning another language adds.
PBR Graphs focus on the physical properties of objects. For example, choosing for a material to reflect light (specular) or to diffuse it and be flat, or for it to be metallic.

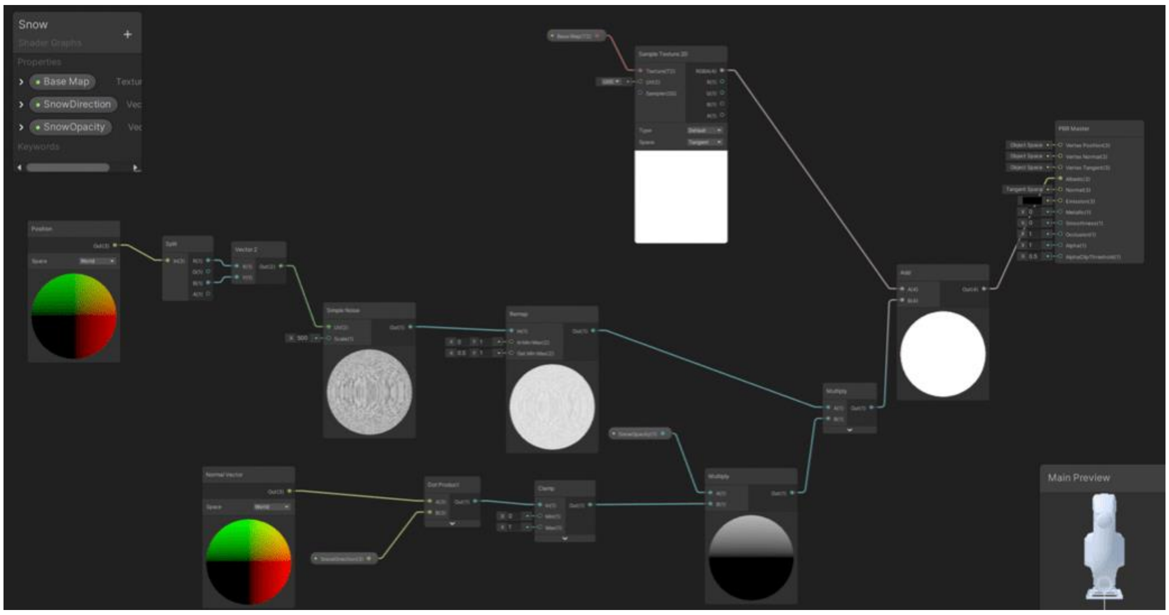


Figure 6 Example of a PBR Graph

3.2.3 Visual Studio

Visual Studio is an IDE developed by Microsoft. Used widely for software, application, and web development. It works with all key programming languages such as C, C++, C# and Java etc, with its code editor and compiler able to execute them all seamlessly.

To create and modify scripts on Unity, I used Visual Studio as my IDE. I had already used this platform for previous University modules relating to C++ and C#, therefore my files automatically opened through Visual Studio anyway.

3.2.4 Photoshop

To realise certain aspects of weather such as clouds and lightning strikes, materials depicting these are required.

Adobe Photoshop is a hugely popular photo editing and manipulation software. It handles large photo files well and has an abundance of tuning capabilities, all spanning over the use of multiple layers. This was the primary choice in creating bespoke assets as I already owned a licence to use the program.

Because of this, I created several custom materials and easily implemented them into the Unity project.

3.3 Development

3.3.1 Preliminary particle systems

Since Unity is a popular development platform, there are a plethora of online resources and tutorials that aid in learning. I used this to my advantage and learnt how to traverse around the system in this way.

Particle systems are a key game object in Unity where one can simulate smoke, clouds, flames and other visual effects by modifying the properties [20]. This will be the primary component in this dissertation because of its versatility and ability to simulate different weather conditions.

From the project plan outlined earlier within Section 1, note how the first implementation duration is double the length of the other subsequent conditions? This is because I get more accustomed to Unity and its interface that the time to create subsequent systems decrease exponentially.

At first, I followed YouTube tutorials to understand particle systems. In doing this, I further familiarised myself with Unity's UI. The first resource I used was Gabriel Aguiar Prod.'s "Unity 2017 – Game VFX – Realistic Rain Effect" [21]. I learnt and understood how this would emit particles that simulate rain, despite being very simple. Because this was my preliminary attempt at using the platform, I applied the system onto a very simple plane with a ball in the centre to test how collisions worked.

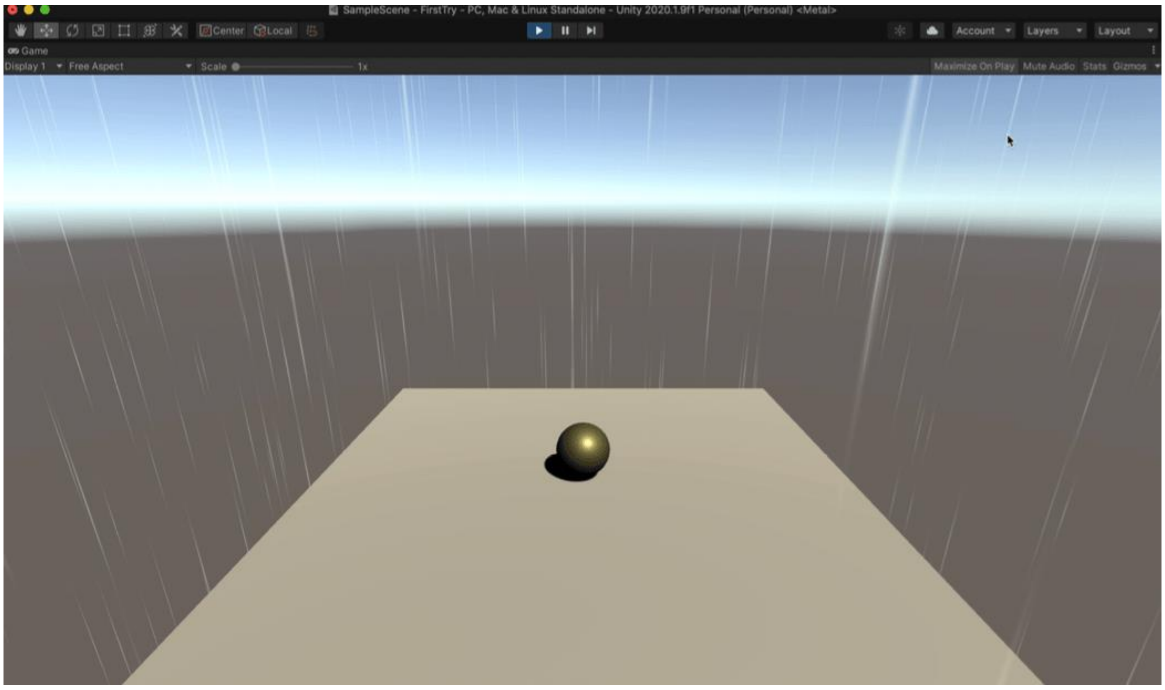

SampleScene - FirstTry - PC, Mac & Linux Standalone - Unity 2020.1.9f1 Personal (Personal) <Metal>
Center
Local
Account
Layers
Layout
Game
Display 1
Free Aspect
Scale
1x
Maximize On Play
Mute Audio
Stats
Gizmos

Figure 7 First try at producing rain using particle systems.

I found that the tutorial was immensely helpful since it resulted in the desired output shown in Fig. 5 above. Difficult to see from a screenshot, but the rain collided with the terrain and didn't go through the plane.

I then followed Mirza's YouTube tutorial titled "Unity VFX – Realistic Snow (Particle System Tutorial)" [22] published in 2016. This applied the same concepts as the rain but by adjusting the properties I could mimic another condition.

Despite these videos being slightly dated, Unity's patch releases have been very inclusive and I could still employ most, if not all of the methods used in these tutorials towards my project.

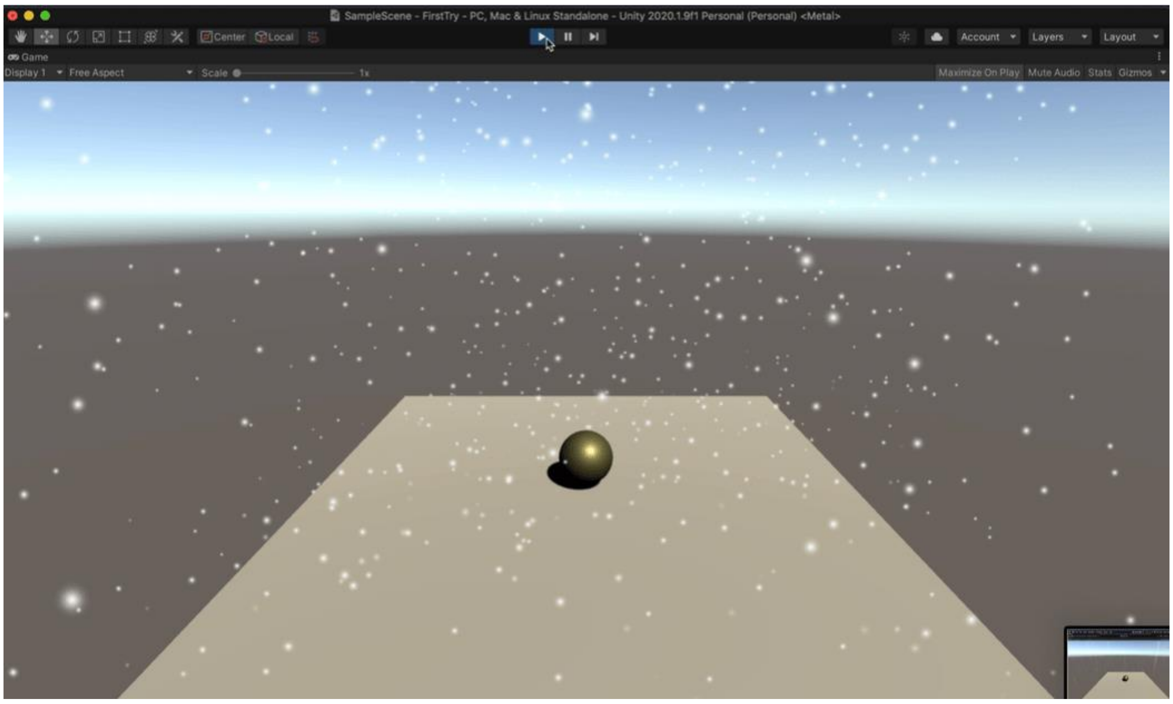

SampleScene - FirstTry - PC, Mac & Linux Standalone - Unity 2020.1.9f1 Personal (Personal) <Metal>
Center
Local
Account
Layers
Layout
Game
Display 1
Free Aspect
Scale
1x
Maximize On Play
Mute Audio
Stats
Gizmos

Figure 8 First try at producing snow using particle systems.

3.3.2 Additional particles for rain

To further my rain particle system, I decided to have ripples and splashes when each particle collides with the terrain. I continued to use Gabriel Aguiar Prod.'s tutorial, and this time focused on their creation of ripples and splashes.

This applied the same concepts as a standard particle system, only that it doesn't emit over time nor distance, but in bursts. This works because it is placed as a child of the rain system in terms of hierarchy, and is set as a sub emitter where it only shows when the rain droplets collide with the plane.

3.3.3 Snow build shader & Pipeline issues

Now that I had a base particle system to work with, the next step would be to implement the resulting effects that these weather conditions have on their environment. For example, this would be snow covering the landscape per the intensity of the snowfall.

Brackeys tutorial on "SNOW in unity – SHADER GRAPH" [23] gave me some insight into the first method I could use. However, his method requires the project to be in HDRP because PBR graphs can only be used within this pipeline.

This proved to be an issue because the tutorials I followed had their projects set to URP (or LWRP then).

Because of this, I created a new project to attempt this new tutorial. In this instance, I used free assets ("LowPoly Environment Pack" by Korveen [24]) from the Unity Asset Store to quickly make an environment that the shader graph can be applied to. Because their assets weren't tuned for HDRP, every component was pink. Materials are coloured pink because this is Unity’s default pink unlit shader which is displayed when a shader is broken. [25] This occurs when upgrading existing projects that are set to a different pipeline into HDRP; or when materials from the Asset Store are not compatible with HDRP shaders.

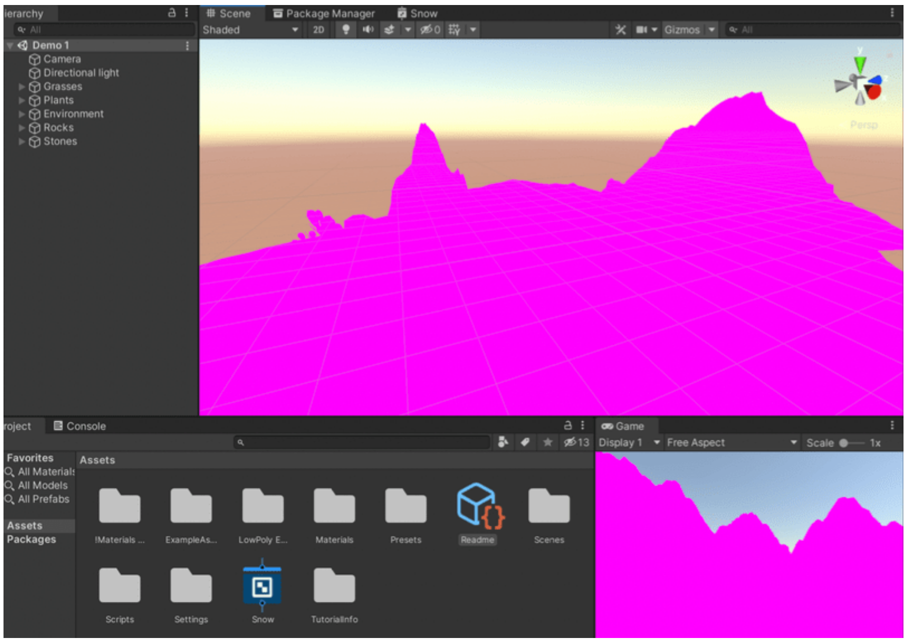


Figure 9 Unity's unlit shader over my non-HDRP compatible assets

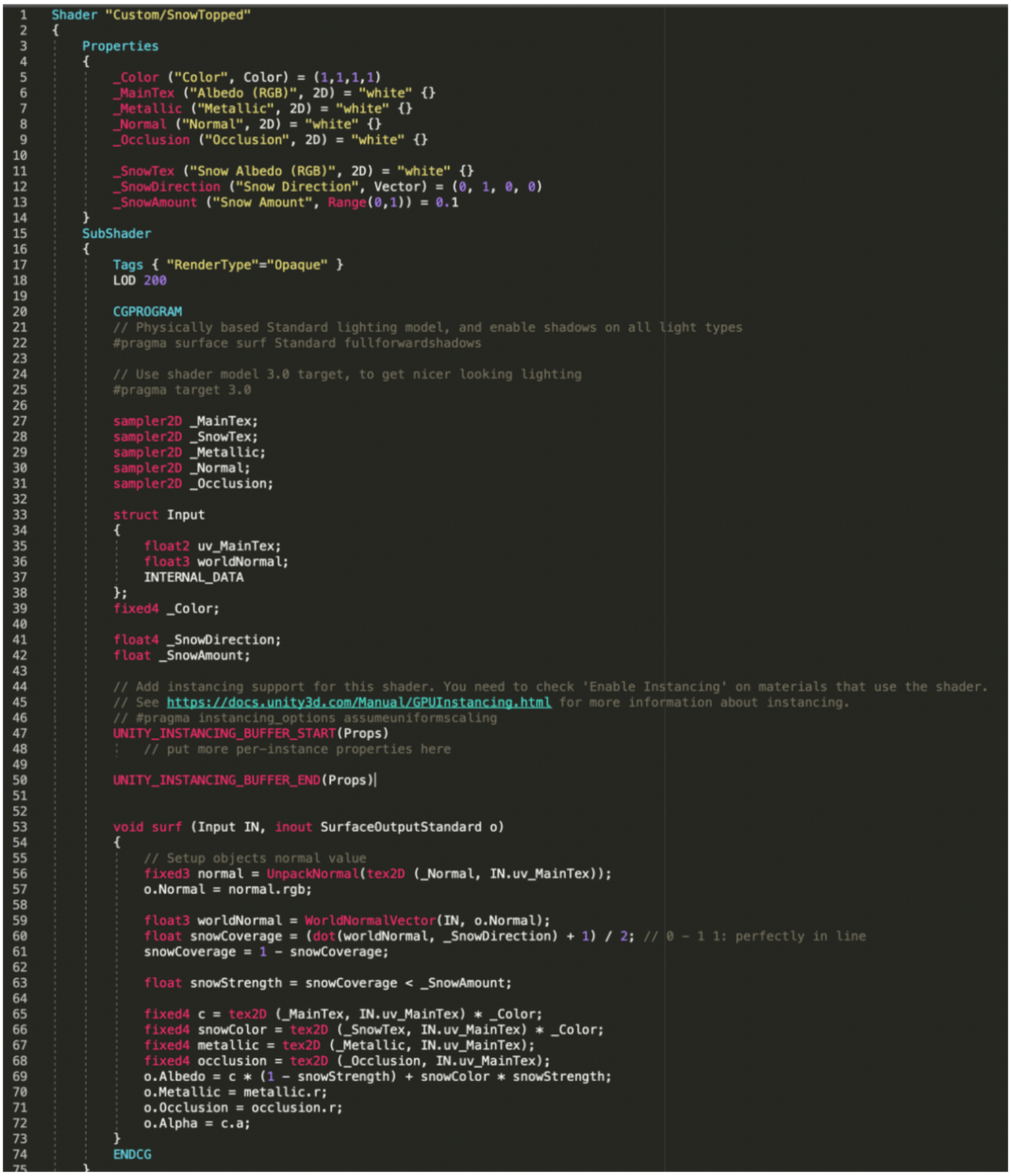
To counteract this, I would have to create a game world with a custom texture over the plane. I decided it was unnecessary to focus my time on making my own because of the time constraint, on top of the project scope not including the creation of my own assets.

On top of this, the snow-topped shader had to be applied to every object in the game which proved to be very tedious; and whilst it could be done with minimal objects on the game world, all the assets were pink anyway so it was difficult to discern between the objects.

I decided that there had to be another easier way to achieve this effect. From my prior research into Unity’s pipelines, the final reason why I chose to continue the project in URP is that I could employ a snow cover effect on the terrain–of the same magnitude as one could achieve through PBR graphs–through scripting.

Following another YouTube tutorial by World of Zero titled “Writing a Snow Covered Shader,” [26] I finalised the code for showing the snow settling.

This tutorial followed the same concepts Brackeys video did, but without the use of PBR Graphs and instead used solid code. This uses a normal map to define the areas where the snow should settle since it contains information on the surface’s details. Making use of the vertices within an object to calculate where the surface directly faces up within the world space. This means that it procedurally generates snow and places it on top of any object within the scene. This also includes a slider that determines how much snow should cover the object.

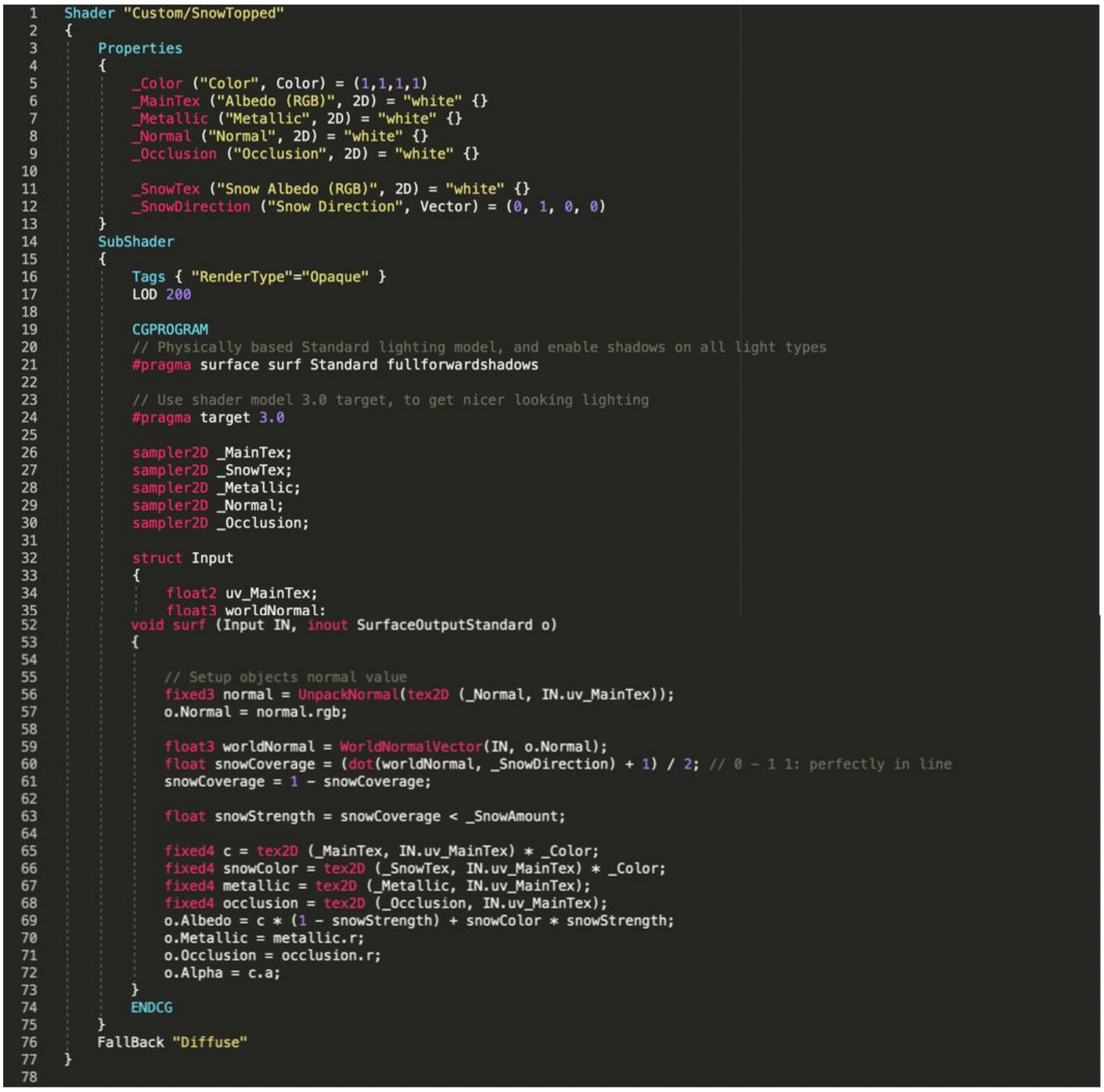


```
Shader "Custom/SnowTopped"
{
    Properties
    {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _Metallic ("Metallic", 2D) = "white" {}
        _Normal ("Normal", 2D) = "white" {}
        _Occlusion ("Occlusion", 2D) = "white" {}

        _SnowTex ("Snow Albedo (RGB)", 2D) = "white" {}
        _SnowDirection ("Snow Direction", Vector) = (0, 1, 0, 0)
        _SnowAmount ("Snow Amount", Range(0,1)) = 0.1
    }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
        // Physically based Standard lighting model, and enable shadows on all light types
        #pragma surface surf Standard fullforwardshadows

        // Use shader model 3.0 target, to get nicer looking lighting
        #pragma target 3.0

        sampler2D _MainTex;
        sampler2D _SnowTex;
        sampler2D _Metallic;
        sampler2D _Normal;
        sampler2D _Occlusion;

        struct Input
        {
            float2 uv_MainTex;
            float3 worldNormal;
            INTERNAL_DATA
        };
        fixed4 _Color;

        float4 _SnowDirection;
        float _SnowAmount;

        // Add instancing support for this shader. You need to check 'Enable Instancing' on materials that use the shader.
        // See https://docs.unity3d.com/Manual/GPUInstancing.html for more information about instancing.
        // #pragma instancing_options assumeuniformscaling
        UNITY_INSTANCING_BUFFER_START(Props)
            // put more per-instance properties here

        UNITY_INSTANCING_BUFFER_END(Props)


        void surf (Input IN, inout SurfaceOutputStandard o)
        {
            // Setup objects normal value
            fixed3 normal = UnpackNormal(tex2D (_Normal, IN.uv_MainTex));
            o.Normal = normal.rgb;

            float3 worldNormal = WorldNormalVector(IN, o.Normal);
            float snowCoverage = (dot(worldNormal, _SnowDirection) + 1) / 2; // 0 - 1 1: perfectly in line
            snowCoverage = 1 - snowCoverage;

            float snowStrength = snowCoverage < _SnowAmount;

            fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            fixed4 snowColor = tex2D (_SnowTex, IN.uv_MainTex) * _Color;
            fixed4 metallic = tex2D (_Metallic, IN.uv_MainTex);
            fixed4 occlusion = tex2D (_Occlusion, IN.uv_MainTex);
            o.Albedo = c * (1 - snowStrength) + snowColor * snowStrength;
            o.Metallic = metallic.r;
            o.Occlusion = occlusion.r;
            o.Alpha = c.a;
        }
        ENDCG
    }
```

Figure 10 Initial SnowTopped shader script

Within the code is a value called '_SnowAmount' that measures how much snow will build upon a specific area. Right now, it is in the form of a slider, and this would be perfect for a project where the sole purpose is to showcase a world demo. However, I need the shader script to build the snow procedurally, building over time instead of manually choosing the amount.

To do this, I followed another tutorial by user b3agz titled "LET IT SNOW – Super Simple Snow System for Unity" [27]. b3agz also applied the same logic as the previous snow shader tutorials, but they include a separate script to build the snow automatically.



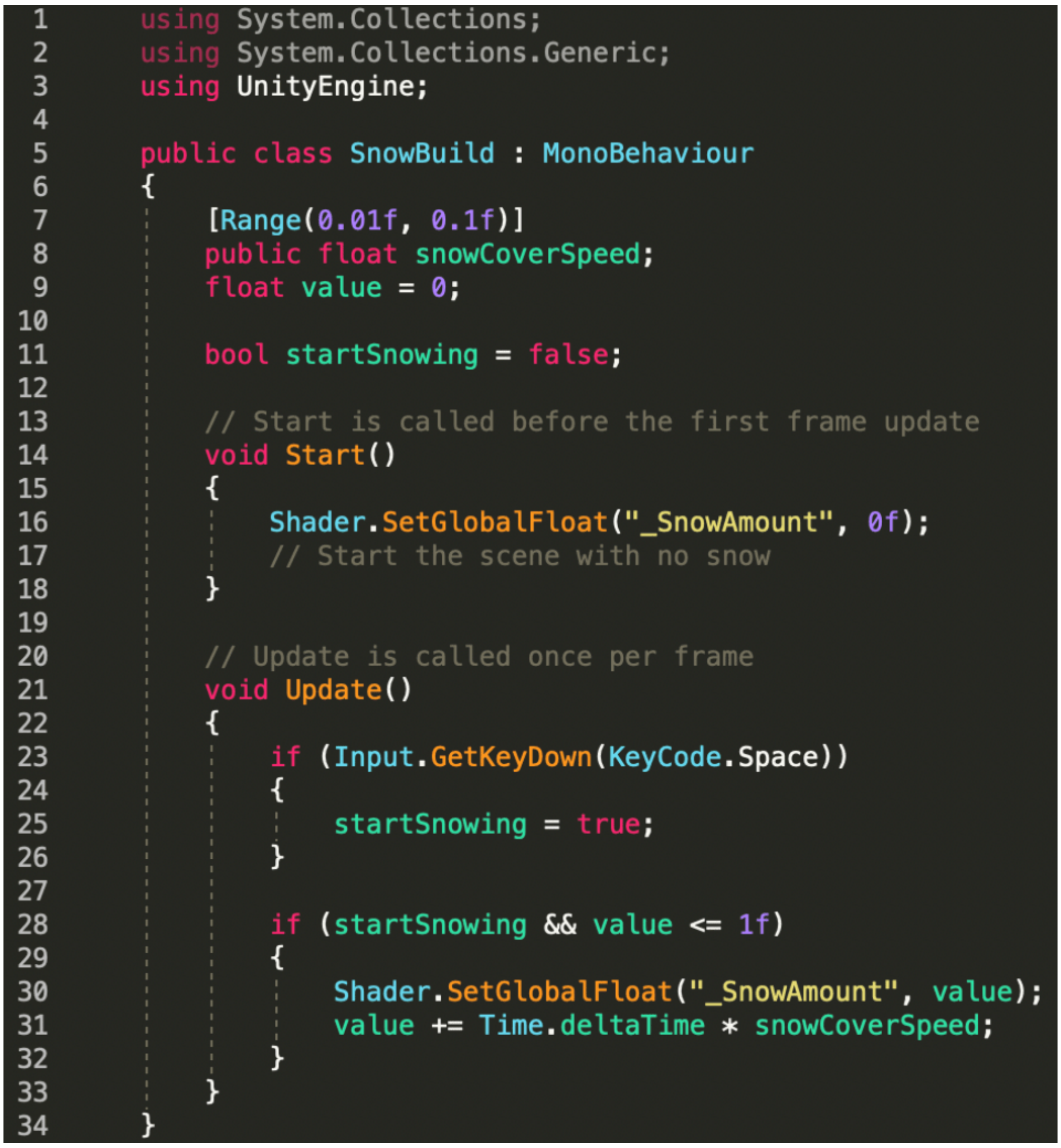
```
Shader "Custom/SnowTopped"
{
    Properties
    {
        _Color ("Color", Color) = (1,1,1,1)
        _MainTex ("Albedo (RGB)", 2D) = "white" {}
        _Metallic ("Metallic", 2D) = "white" {}
        _Normal ("Normal", 2D) = "white" {}
        _Occlusion ("Occlusion", 2D) = "white" {}

        _SnowTex ("Snow Albedo (RGB)", 2D) = "white" {}
        _SnowDirection ("Snow Direction", Vector) = (0, 1, 0, 0)
    }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
        // Physically based Standard lighting model, and enable shadows on all light types
        #pragma surface surf Standard fullforwardshadows

        // Use shader model 3.0 target, to get nicer looking lighting
        #pragma target 3.0

        sampler2D _MainTex;
        sampler2D _SnowTex;
        sampler2D _Metallic;
        sampler2D _Normal;
        sampler2D _Occlusion;

        struct Input
        {
            float2 uv_MainTex;
            float3 worldNormal;
        void surf (Input IN, inout SurfaceOutputStandard o)
        {

            // Setup objects normal value
            fixed3 normal = UnpackNormal(tex2D (_Normal, IN.uv_MainTex));
            o.Normal = normal.rgb;

            float3 worldNormal = WorldNormalVector(IN, o.Normal);
            float snowCoverage = (dot(worldNormal, _SnowDirection) + 1) / 2; // 0 - 1 1: perfectly in line
            snowCoverage = 1 - snowCoverage;

            float snowStrength = snowCoverage < _SnowAmount;

            fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
            fixed4 snowColor = tex2D (_SnowTex, IN.uv_MainTex) * _Color;
            fixed4 metallic = tex2D (_Metallic, IN.uv_MainTex);
            fixed4 occlusion = tex2D (_Occlusion, IN.uv_MainTex);
            o.Albedo = c * (1 - snowStrength) + snowColor * snowStrength;
            o.Metallic = metallic.r;
            o.Occlusion = occlusion.r;
            o.Alpha = c.a;
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

Figure 11 Modified SnowTopped shader script

Instead of creating a new shader script, I altered the SnowTopped shader from before and removed '_SnowAmount' from the properties box as that would have overridden the script for procedurally building after.

I then created a new script called SnowBuild to increase the snow cover level as time went on. If the value of the snow is less than 1 (the whole environment is covered with snow), and if the Boolean value 'startSnowing' is true, the script would run and would update the snow amount per frame that passes.

This is possible because the Update function is called once per frame, and within that function 'snowAmount's value is set to the number of seconds it took for the project to process the previous frame, multiplied by the speed set by the user (inside 'snowCoverSpeed'). In turn, this will procedurally increase the amount of snow on each object since the variable 'value' will be increasing.



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

public class SnowBuild : MonoBehaviour
{
    [Range(0.01f, 0.1f)]
    public float snowCoverSpeed;
    float value = 0;

    bool startSnowing = false;

    // Start is called before the first frame update
    void Start()
    {
        Shader.SetGlobalFloat("_SnowAmount", 0f);
        // Start the scene with no snow
    }

    // Update is called once per frame
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            startSnowing = true;
        }

        if (startSnowing && value <= 1f)
        {
            Shader.SetGlobalFloat("_SnowAmount", value);
            value += Time.deltaTime * snowCoverSpeed;
        }
    }
}
```

Figure 12 SnowBuild script

The snowfall is quite fast and doesn't build with the same intensity and speed as the particle system. To fix this, I changed the 'snowCoverSpeed' from a range between 0.01-0.1 to a fixed number of 0.001. This is because I found no use in having the option to choose between two constants if the snowfall is only going to fall at a certain speed.

Additionally, the snowfall does not correlate to the particle system's snow specks reaching the terrain, so right now snow appears on the ground whilst the particles have yet to touch it. To fix this, I used Unity's documentation on the WaitForSeconds class [28]. I revised the StartSnow function from a standard public function into an IEnumerator type. This allowed me to use a coroutine, and subsequently the WaitForSeconds class. I calculated the number of seconds it took for the snow to reach the ground and have the snow build on the environment naturally, and it resulted in an 8-second wait.

3.3.4 Procedural mesh generation attempt

A consistent game world is essential to portraying weather effects, so the particles have a platform to influence. I ventured into learning how to create scripts by developing a procedural mesh generator using another YouTube tutorial titled "PROCEDURAL TERRAIN in Unity! – Mesh Generation" by Brackeys [29].

Following this produced a MeshGenerator class that procedurally generated a mesh using different triangles. This was successful in producing a different mesh every time. I then encountered an issue where the particles from the snow/rain particle system went through the mesh—despite having a mesh collider component added onto the object.

Eventually, I couldn't find the answer to the issue and abandoned this endeavour.

Whilst I decided against using a procedural mesh to generate the environment, I found that this was necessary for getting experience in writing scripts in C#, and taught me a lot about loops, functions and getting components directly from the project.

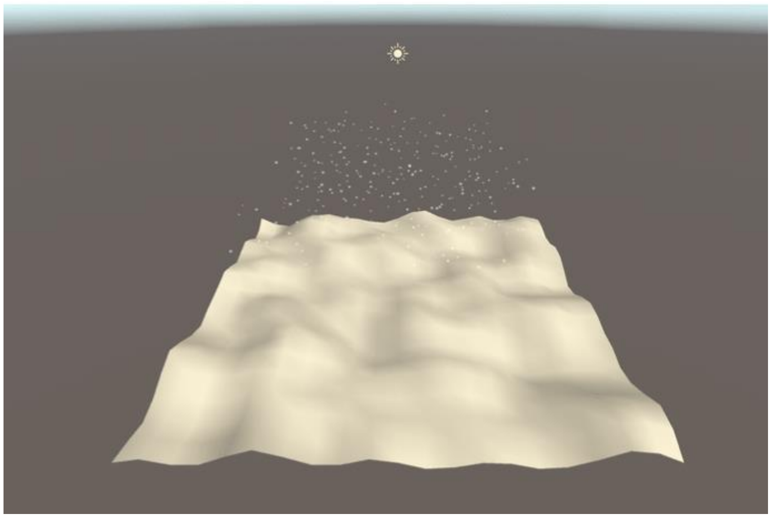


Figure 13 Procedural mesh generation example

3.3.5 Finalising game world

Next, I tried another free asset named "Mountain Terrain + Rock + Tree" made by Creative Poly [30], to which I used the mountain as the main terrain. This was a great addition as it contained 4 texture maps: albedo transparency (the base colour of the object), ambient occlusion (simulates the shadows made by environmental effects), normal map (adding surface detail onto the texture), and metallic smoothness (controlling the reflections).

Again, this asset also had the same problem of the particles going through the prefab (short for prefabrication: a premade asset. In this instance, the mountain already had the texture maps applied to it with the appropriate parameters). But applying a mesh collider with the same properties like the one I attempted to apply to the procedural mesh lead to it working with this mountain.

Using this mountain allowed me to learn how mesh colliders worked, but I was still unsatisfied with this asset and wanted a terrain that had more defined peaks and valleys. I found that none of the free assets on the Unity Store met my vision for the project, so I decided to create my own terrain.

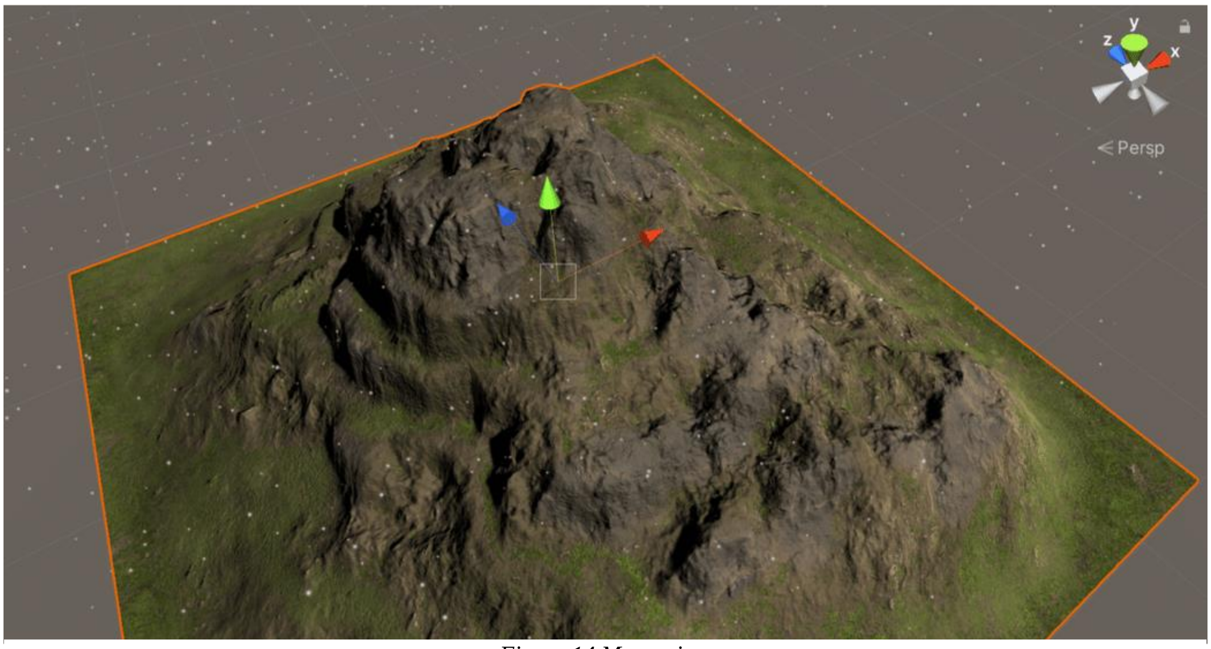


Figure 14 Mountain asset

This is possible using Unity's built-in terrain tools. I followed Brackey's YouTube tutorial titled: "How to make Terrain in Unity!", [31] which thoroughly explained the components. Through the terrain tools, I could modify and manipulate a flat plane using several brushes to create hills and dips. Moreover, the settings also allowed me to paint over the blank terrain using different textures, most of which are pre-defined by the platform such as grass, stone etc. I also added a terrain collider component onto the game object which allows the terrain to be affected by particle systems.

I found that creating a custom terrain was great in terms of having a customised environment, however, it doesn't allow for separate texture maps. To counteract this, I applied the same material from the mountain asset (which had opportunities to include maps for normals, occlusion, metallic etc.) to the new terrain. This means that I can apply scripts that use these map settings.

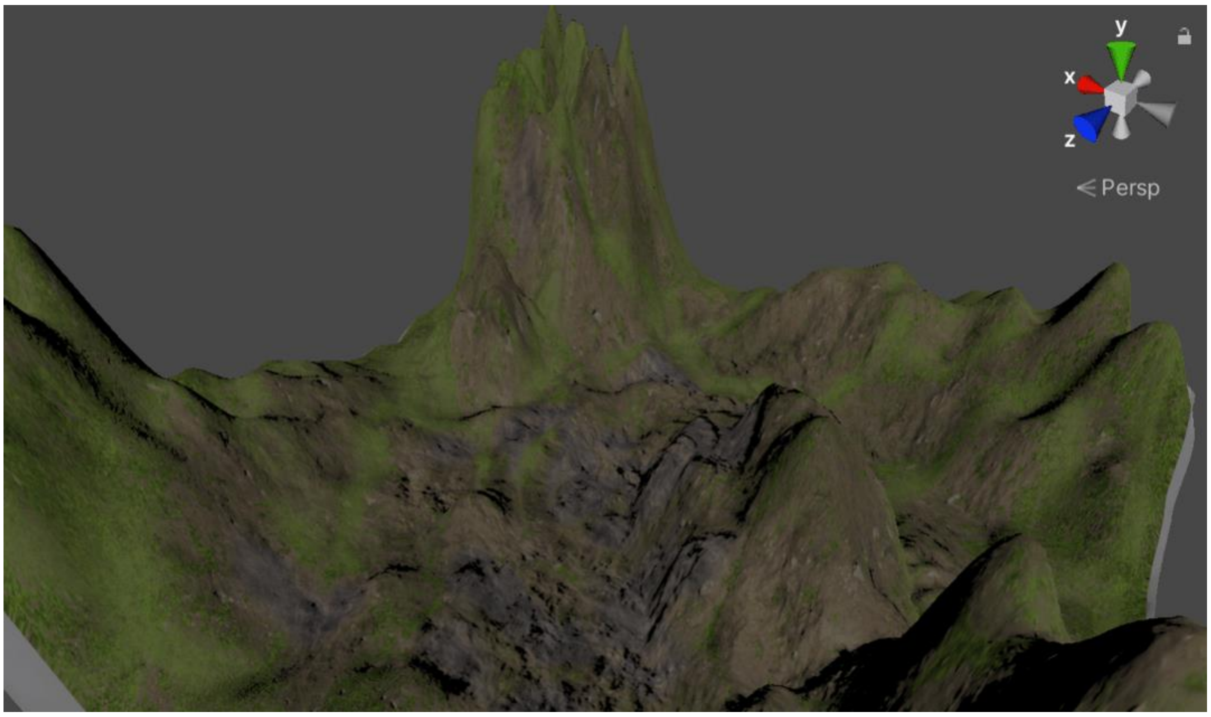


Figure 15 Final terrain

3.3.6 Cloud particle system

The next step was to create clouds to add to the snow scene. At first, I created my own cloud material on photoshop following Gabriel Aguiar Prod.'s tutorial: "Unity 2017 – Game VFX – Realistic Rain Effect" again. This video also covers how they create clouds to accompany the weather effects.

To do this I used free Photoshop cloud brushes available online, produced by Mila Vasileva [32]. After, I imported it into my scene and created a new particle system afterwards similar to ones of rain and snow, only that it emits in the same section and doesn't fall.

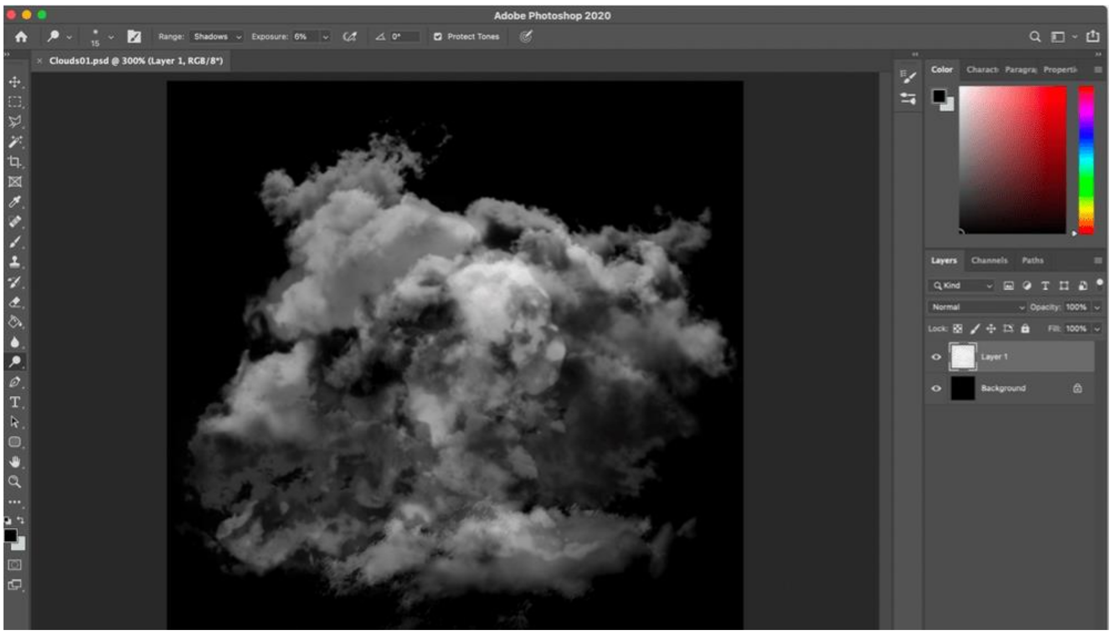


Figure 16 Custom cloud texture in Photoshop

3.3.7 Thunder

For thunder, I followed Gabriel Aguiar Prod.'s tutorial titled "Unity 2017 – Game VFX - Lightning and Thunder Effect". His tutorials proved immensely helpful since he covers a wide range of rendering topics. Again, this required me to create another material of my own through photoshop.

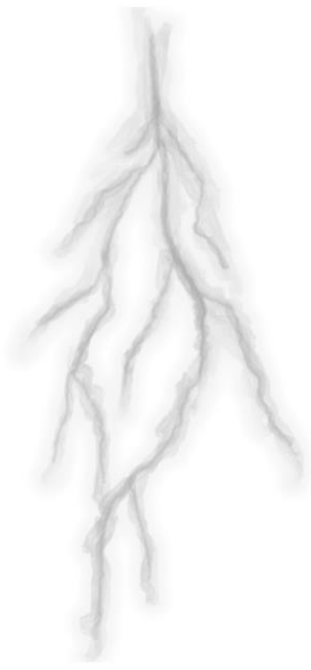


Figure 17 Lightning material

Figure 18 Storm cloud material

Since I've already experienced the process of adding custom materials from the previous addition of clouds, the procedure was easy. This time the particle system isn't a sub emitter like rain splashes and ripples, but its own standalone one that emits once every second.

Lightning is employed for the storm counterpart of rain. This system would be identical to rain but additionally have darker clouds and lightning. Upon creation of another set of custom cloud materials, seen in Figure 18, I found that there were no discernible visual differences between the normal and storm clouds when the project was run. Because of this, I just used the existing clouds material for the storm system.

3.3.8 Fog addition

With the addition of fog, this completes the weather conditions I aimed to implement at the start. Before the development of this system, I researched a few ways to recreate fog on Unity. A notable source of information was a video by TheSenpaiCode on YouTube titled: "Unity3d, Different Ways To Make Fog With Lighting & particle Systems". [33] In this video, the author goes through two ways of making fog: The first is by using Unity's built-in fog settings under the game lighting options. These options include changing the colour of the fog, the rendering mode, and the fog density. While this would be the easiest option for a developer, this does not demonstrate much of the proposed, manual rendering techniques that I expect to present in this project. Moreover, the effect this fog preset has on the world is to replace the texture of every game object with a white (or the chosen colour) film that gradually regains its original material the closer the object is to the player. I feel this setting is more suited for large scale environments where the white wash over the terrain can be more realistic because the player would not be able to see the edge of the world. This effect can be seen in Figure 19 below, note the white parts of the ground stops abruptly.

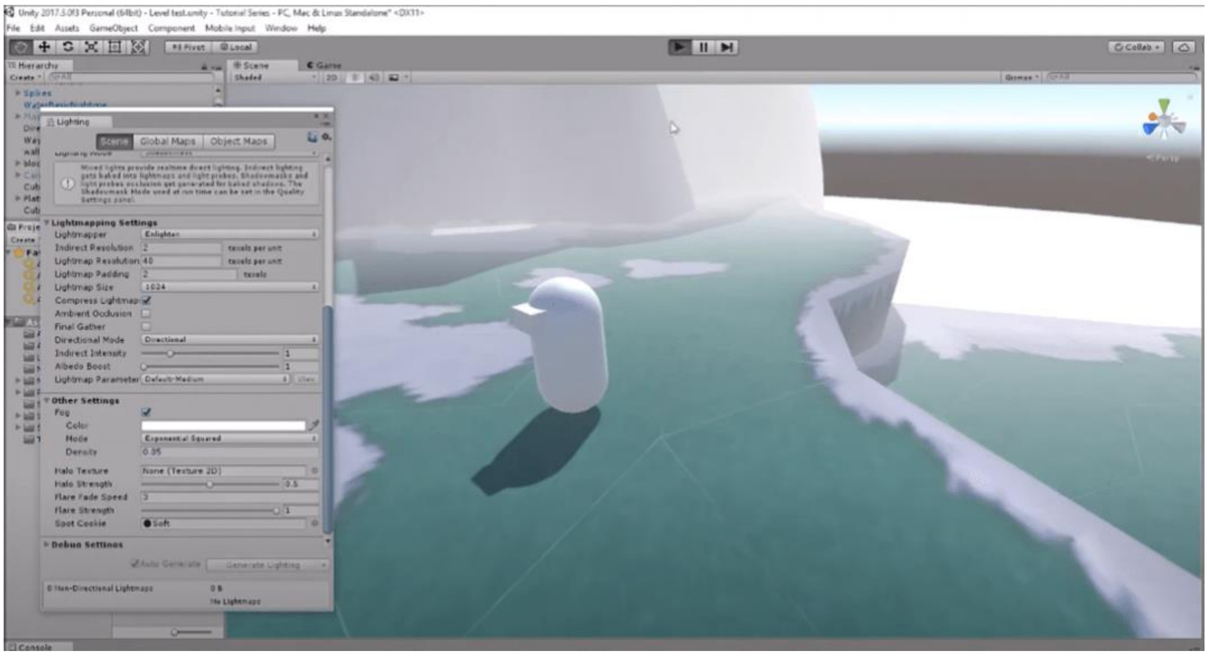


Figure 19 Screenshot of TheSenpaiCode's video, showing Unity's built-in fog effects

Since my project's environment is similar to the example shown above, in the sense that it is simply a flat plane that does not span as far as the user's camera can see, I decided against this method.

The second method shown in the video is the same process as implementing lightning and clouds in terms of creating the custom material on photoshop and then importing it into Unity. Again, I used free Photoshop fog brushes available online on Brusheezy [34] and created the final particle system. This has the same properties as the clouds system, only it emits closer to the ground of the game world.

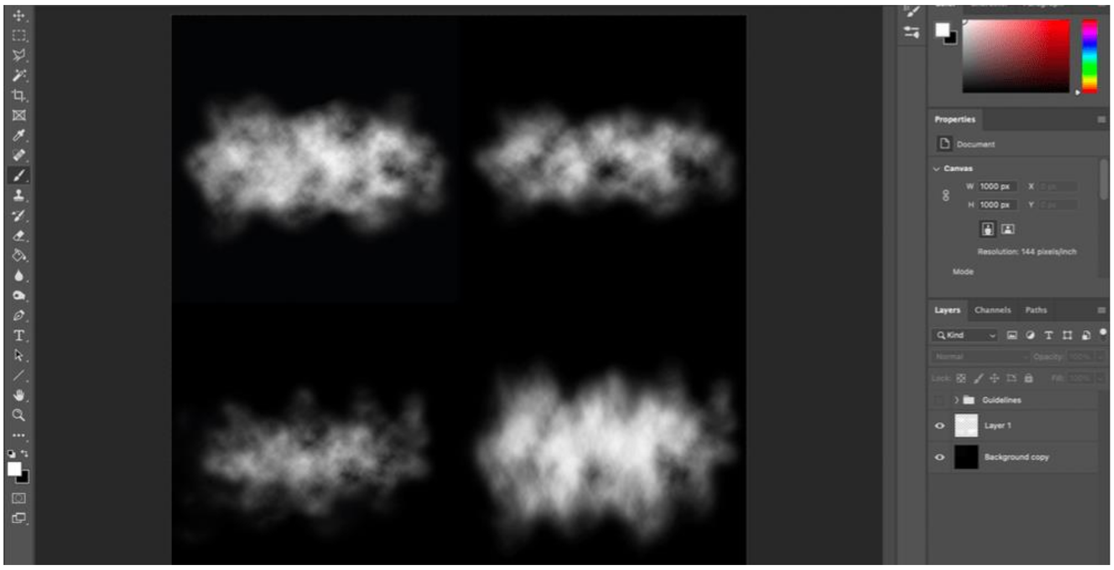


Figure 20 Custom fog texture in Photoshop

3.3.9 User interface design

As of this developmental phase, the weather conditions were selected and shown manually through the deletion of the other particle systems. For example, if I wanted to show rain I would have to delete any game objects relating to snow, and then undo the changes after I was finished with viewing the rain effect, and vice versa.

Instead, I implemented a simple user interface (UI) using the canvas game object (a static area where one could place objects, always seen on the players' screen) and buttons to activate each system. “Buttons are an essential component of an interaction design, playing a major role in delivering an action” [35] and I also thought they were the natural choice.

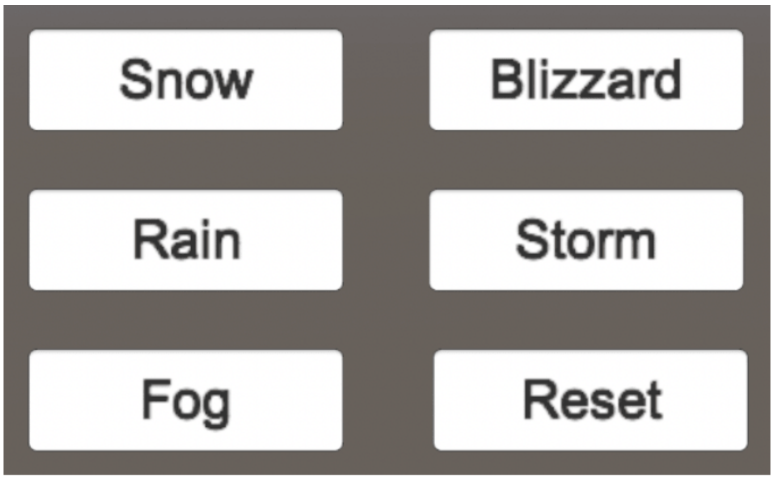


Figure 21 Weather condition buttons
(fog, blizzard, storm and reset buttons currently placeholders)

To properly apply this design, the snow script would have to retrieve all the components within the project so each system can be enabled and disabled. This means that once a button is pressed, that specific condition will begin and also deactivate any current ones. For example, SnowBuild is now renamed as SnowScript and has a function called BeginSnowing which takes in multiple particle systems and assigns their emission module to another variable that can be enabled and disabled. This is seen in Figure 22.

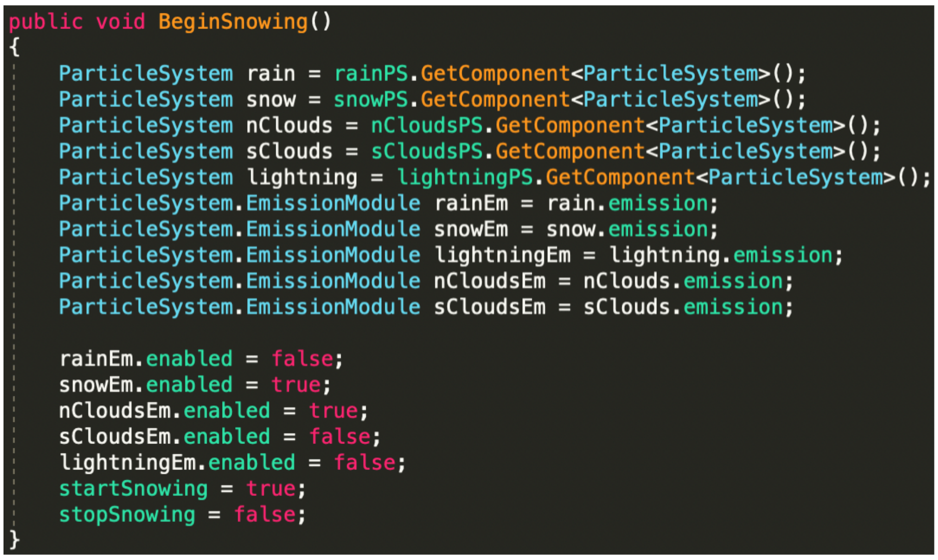

```
public void BeginSnowing()
{
    ParticleSystem rain = rainPS.GetComponent<ParticleSystem>();
    ParticleSystem snow = snowPS.GetComponent<ParticleSystem>();
    ParticleSystem nClouds = nCloudsPS.GetComponent<ParticleSystem>();
    ParticleSystem sClouds = sCloudsPS.GetComponent<ParticleSystem>();
    ParticleSystem lightning = lightningPS.GetComponent<ParticleSystem>();
    ParticleSystem.EmissionModule rainEm = rain.emission;
    ParticleSystem.EmissionModule snowEm = snow.emission;
    ParticleSystem.EmissionModule lightningEm = lightning.emission;
    ParticleSystem.EmissionModule nCloudsEm = nClouds.emission;
    ParticleSystem.EmissionModule sCloudsEm = sClouds.emission;

    rainEm.enabled = false;
    snowEm.enabled = true;
    nCloudsEm.enabled = true;
    sCloudsEm.enabled = false;
    lightningEm.enabled = false;
    startSnowing = true;
    stopSnowing = false;
}
```

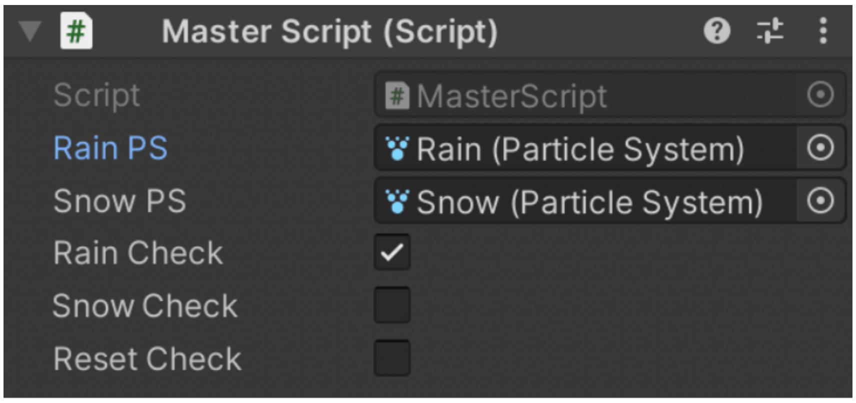
Figure 22 BeginSnowing function, showing component retrieval

3.3.10 Scripting changes and reset button

At first, I had a separate script for each system i.e. RainScript, FogScript. However, I felt it could be consolidated into one master script to reduce computational costs. Upon further research, it seems it does not affect runtime [36], but I found it was more streamlined to have one overall script that I can apply changes to.

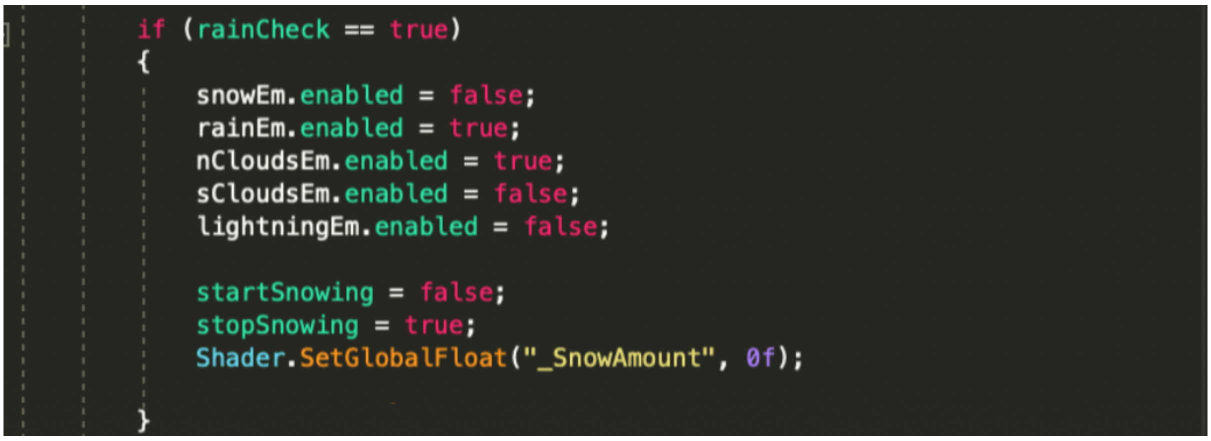
So, I combined them all into a MasterScript which has the same logic as the SnowScript in taking in all the components in the project and selecting which to enable. Except this additionally has public checkboxes that I can enable within the properties pane of Unity. And through this, I have multiple if statements to differentiate the particle systems that need to be enabled.

I then created 4 different instances (empty game object) on Unity and named them after each weather condition. Within these, I attached the MasterScript onto them, selected the correct particle systems, and pressed on the correct checkbox.



Master Script (Script)	
Script	MasterScript
Rain PS	Rain (Particle System)
Snow PS	Snow (Particle System)
Rain Check	✓
Snow Check	
Reset Check	

Figure 23 The public variables within MasterScript, note the checkboxes at the bottom that discern each system from one another



```
if (rainCheck == true)
{
    snowEm.enabled = false;
    rainEm.enabled = true;
    nCloudsEm.enabled = true;
    sCloudsEm.enabled = false;
    lightningEm.enabled = false;

    startSnowing = false;
    stopSnowing = true;
    Shader.SetGlobalFloat("_SnowAmount", 0f);

}
```

Figure 24 An example of the checkboxes and if statements being used together

To reset the game world back to its original state of no conditions, I implemented the reset button onto the UI. The process turned out to be straightforward due to the conditions being combined into one script. I simply attached another if statement and set all of the particle systems' emissions to false.

Furthermore, when a weather condition is enabled after another (snow is running, and the user activates rain), it will also call the stopSnowing function. This is a new function that resets the game world texture back to the original by reusing the original snow build code but reducing the value of the snow amount every frame.

3.3.11 Audio implementation

To further achieve realism, the sounds accompanying rain and snow are essential. I found a plethora of free soundbites from YouTube which I will use within the project with proper credits. Rain must have rainfall, and then thunder when the emission of rain is high enough. Whereas snow should have a wind gust effect. I decided to neglect giving fog a sound as I do not feel it produces any sound.

To implement sounds, I first downloaded small sound clips from YouTube for rain: "Nature Sounds: Rain Sounds One Hour for Sleeping, Sleep Aid for Everybody" [37] by MeditationRelaxClub – Sleep Music & Mindfulness. Then "Thunder Sound No Rain" [38], and "Wind Blowing Sound Effect" [39] from Free Sounds Library.

I imported these files onto Unity and proceeded to create empty Game Objects to append them to. I then added the audio sources onto the MasterScript under the correct systems. I learnt how to do this using a forum post titled "How do I use an Audio Source in a script" on Unity Support [40].

I found that the switch between systems was too abrupt due to the sounds being sharply cut off. To fix this, I implemented an audio fade script into the program. I used a tutorial by gamedevbeginner [41] and added a script called FadeAudioSource. The audio fades because the script uses "… a coroutine to Lerp the volume from one value to another over a set duration for an even, linear fade." Lerp means Linear Interpolation and is used to return a value between two points on a scale, this is essential to change the volume of the audio source over a period of time. It takes in the audio and gradually brings the audio to the target volume; 1 for 100% or 0 for silent.

Therefore, instead of just playing the audio sources outright, I stop all the sounds which bring their volume to 0 and then raise the desired sound(s) to 100%.

3.3.12 Skybox change

As of this point, the skybox was set to Unity's default properties. At first, I planned on implementing a sky that changes dynamically when switching between different weather systems. For example, the sky should change to a darker skybox for when the rain becomes more intense to signify thunder clouds.

This plan would have required me to employ the same linear interpolation method from the audio fading earlier, but with two different skyboxes. Although upon further inspection of the assets I found on the Unity Asset Store, I was unable to program a script that made use of the assets available; which were 6 sided skyboxes. The resources I found online to aid me in programming a script were for cubemaps or panoramic skyboxes, and I ultimately decided to discard that idea.

Instead, I used an asset by Key Mouse titled "Customizable skybox" [42] and chose a green-hued skybox that made it easier to view the particle systems.

3.3.13 FPS counter

Although Unity provides the user with an FPS counter within the editor settings, the user cannot see it if the program were to be released. For analysis and performance tracking purposes, I added an FPS counter within the canvas area.

At first, my code would update every millisecond which wasn't useful as it wasn't the desired output. I used code from the Unity Forum page, titled: "FPS Counter" which involved a user asking a question about how to implement a counter. A user named kubajs then submitted a code snippet that allowed me to easily add it to my project. This resulted in my StatsDisplay script.

On top of an FPS counter, I will need to calculate the average FPS while the system runs for a minute during the testing phase. This is because the weather condition needs to fully render to be an accurate depiction of the system. For example, the snow system waits for 8 seconds before building on the terrain and taking the FPS at the start in comparison to when the snow has completely covered the ground will output differing results. So, recording the FPS at only one point would be too inaccurate.

I added onto the existing FPS counter I developed earlier and created a one-dimensional array of size 60 (seconds). Then added the current FPS value into the array and finally displayed all the contents of the array, divided by 60 when the array is full.

Finally, I am also considering the memory usage of each system. I used the same logic as the frames per minute (FPM) counter and instead retrieved the currently allocated memory used. This is possible through the memory profiler module built into Unity. [43]

3.3.14 Emission sliders

In order to test the performance concerning the appearance of the project, I removed the buttons for the systems which had a set emission rate. In its place are sliders in the UI that one can adjust the emission for each system, which relates to how strong the particle system looks on the screen.

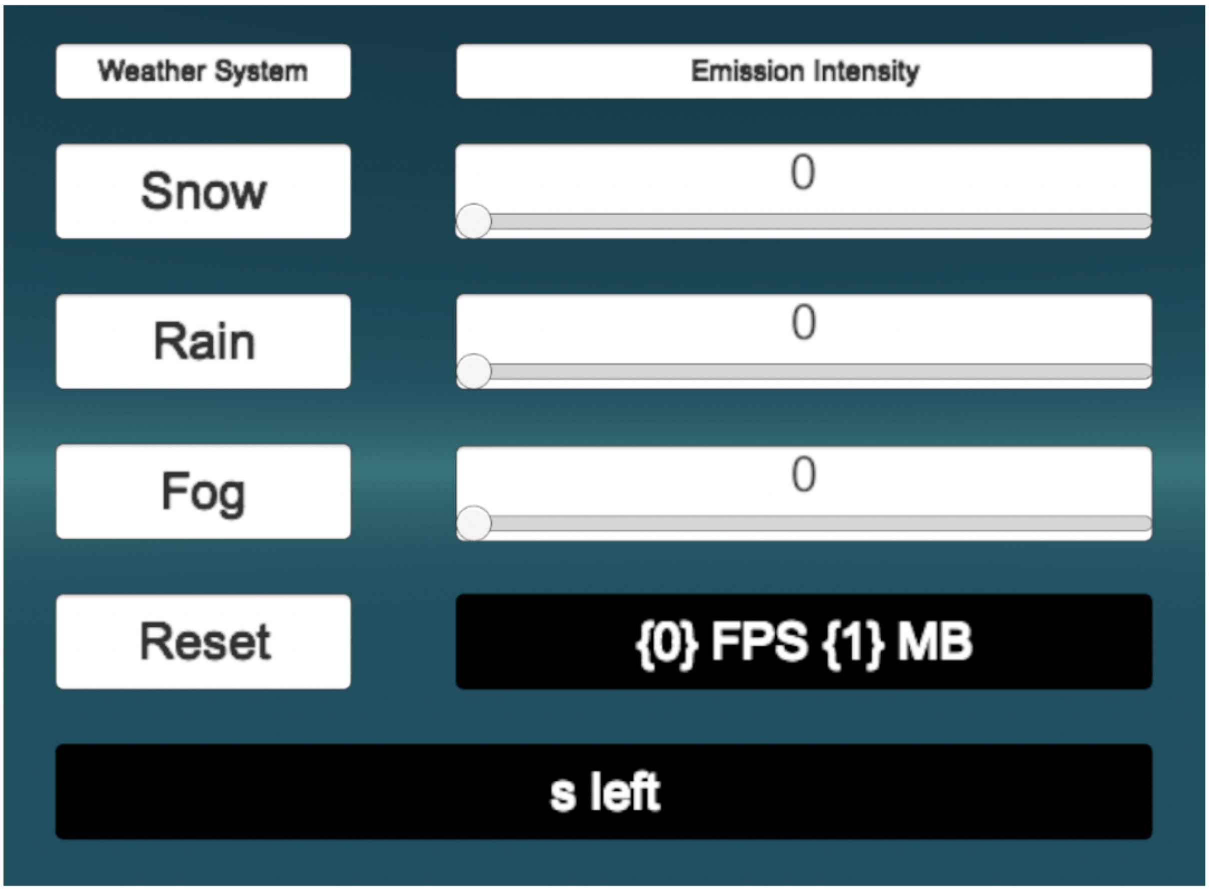


Figure 25 Final UI

To do this, I added the UI slider element onto the canvas area and linked it to the master script. Where the slider value corresponds directly to the emission intensity of the particle system. At 0 emission intensity, the systems do not get called and their sound does not play either.

Because of this, I removed the storm and blizzard button. Thunder and its matching sound will now appear and play when the emission intensity of rain is greater or equal to 250. This is because it is natural for thunder to occur when the rain is most intense.

I have also amended the FPS area to consider the FPS and memory usage. The final black box below everything is for the average FPS and memory, it displays the average value after a countdown from 60.

Finally, the reset button also resets all of the associated FPS and memory values and starts the countdown again from the 60 seconds. I added a function within the StatsDisplay script named “Reset” and it sets all the values to 0 and clears the arrays.

I call this function whenever the user presses on the weather condition buttons because when these are pressed, they consider the emission value from the slider. In other words, this allows me to reset the values whenever I try a new particle system, making testing easier.

3.3.15 Fog performance issues

Unfortunately, the fog causes the project to run extremely slowly. At first, I thought it was due to the custom texture’s file size. But upon downsizing the image from 1000 x 1000px to 250 x 250px, the difference was negligible. Any smaller would cause the fog to be too low quality since it’s a 4 by 4 grid of different fog types. Despite being near identical to the cloud particle system, the performance difference is staggering.

In theory, it should run smoothly since the fog system contains only two systems when enabled; whereas rain has 4 different particle systems and runs at nearly the same FPS as the program at idle. I also amended the Unity rendering settings in hopes it was my computer that caused this steep performance decline. Moreover, I changed the texture quality from full resolution to a quarter and turned off V-Sync. Despite all of these changes, it still results in 3 FPS.

The performance drop was a very interesting discovery because from my prior research into heterogeneous fog by Guo, F. et al., it would seem that my method of generating fog would lead it to be unique in transmission, and easy for the computer to render.

3.3.16 Clearing snow and blizzard system issue

Moving the snow script into the MasterScript made the clearing snow function stop working, and after multiple attempts at simply debugging and amending the code, I was unable to fix the issue. Both the scripts were identical, yet only when the snow was being created in a separate script will the clearing work.

Therefore, I removed the snow section from the master script and re-made the previous SnowBuild script. Now when the reset button or any other system button is pressed, it calls the master script to run the appropriate lines to activate the particle systems, and then also the snow script’s clearSnow function.

3.3.17 Implementation Conclusion

To summarise the implementation progress: I have implemented rain, snow, fog, and a button to reset the changes.

- Rain encompasses a particle system for the raindrops, another for the ripples and small splashes made upon collision on the terrain, and the clouds where the rain is formed. This also includes the sound of rainfall.
  If the user were to choose an emission intensity of greater than or equal to 250, lightning will appear alongside an audio file of thunder clapping.
- Snow includes a particle system for the snowfall, the sound of a snowstorm, and a script to build up snow on the environment.
- Fog just includes a particle system to simulate a fog blanket.
- Finally, the reset button reverts the world to its original state.

3.4 Screenshots of the final build

Figure 26 displays the initial view of the game world when loaded, showing the terrain and its dips and hills. On the top left are the buttons for activating each weather system. Next to the buttons are the emission values on a slider. The black box next to the ‘Reset’ button displays the performance statistics. The final black box below that is the countdown until the program shows the average performance statistics in 60 seconds.

Figure 27 is an alternate view of the terrain, possible through the camera controller.

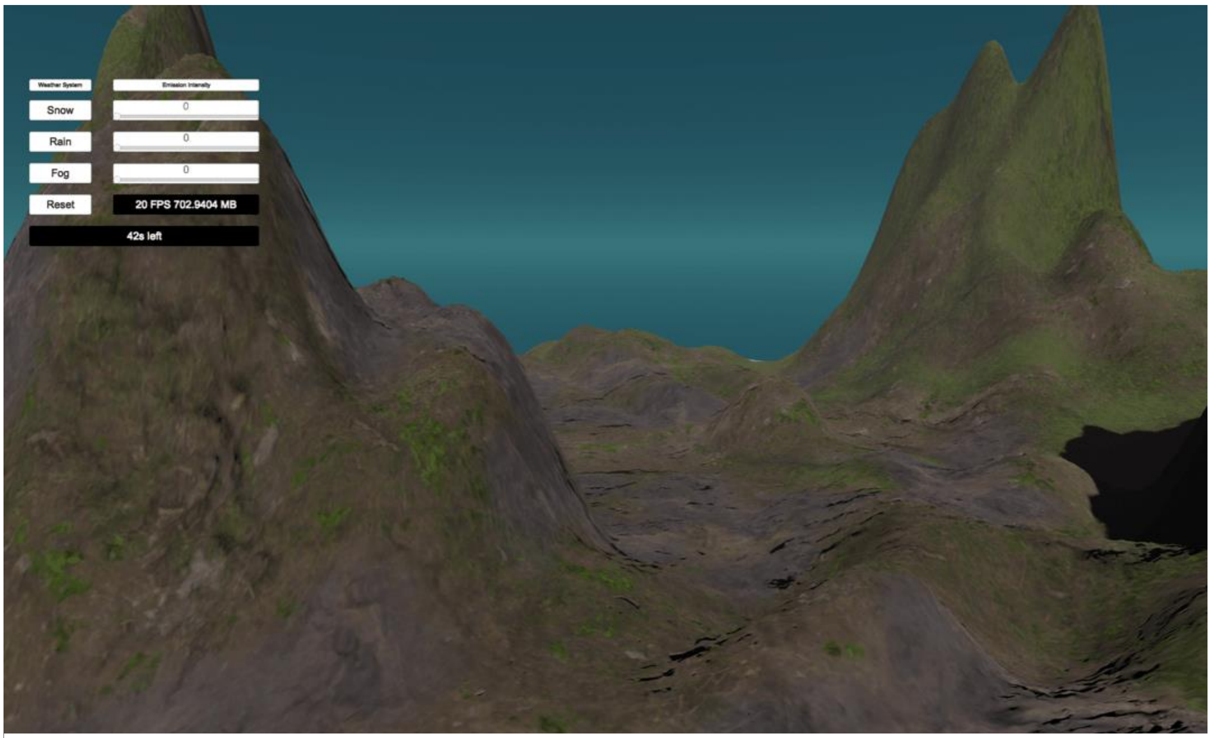


Figure 26 Initial view

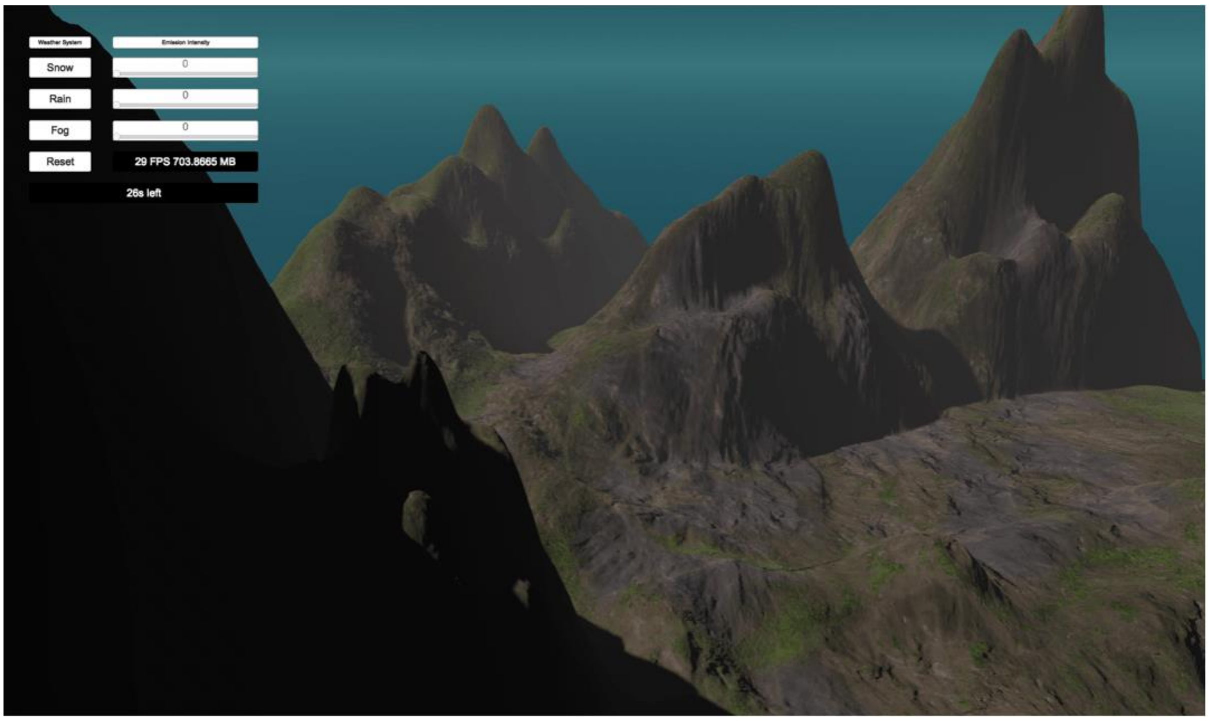


Figure 27 Alternate angle of the environment

Figure 28 and 29 show the snow system in use. The particles are small to be natural-looking and to account for the large terrain, however, you can see the snow particles by zooming into the photos. Figure 30, in particular, is a closer look into the snow building on the environment. Note the sides of the hills are bare, whilst the lower parts of the ground are covered.

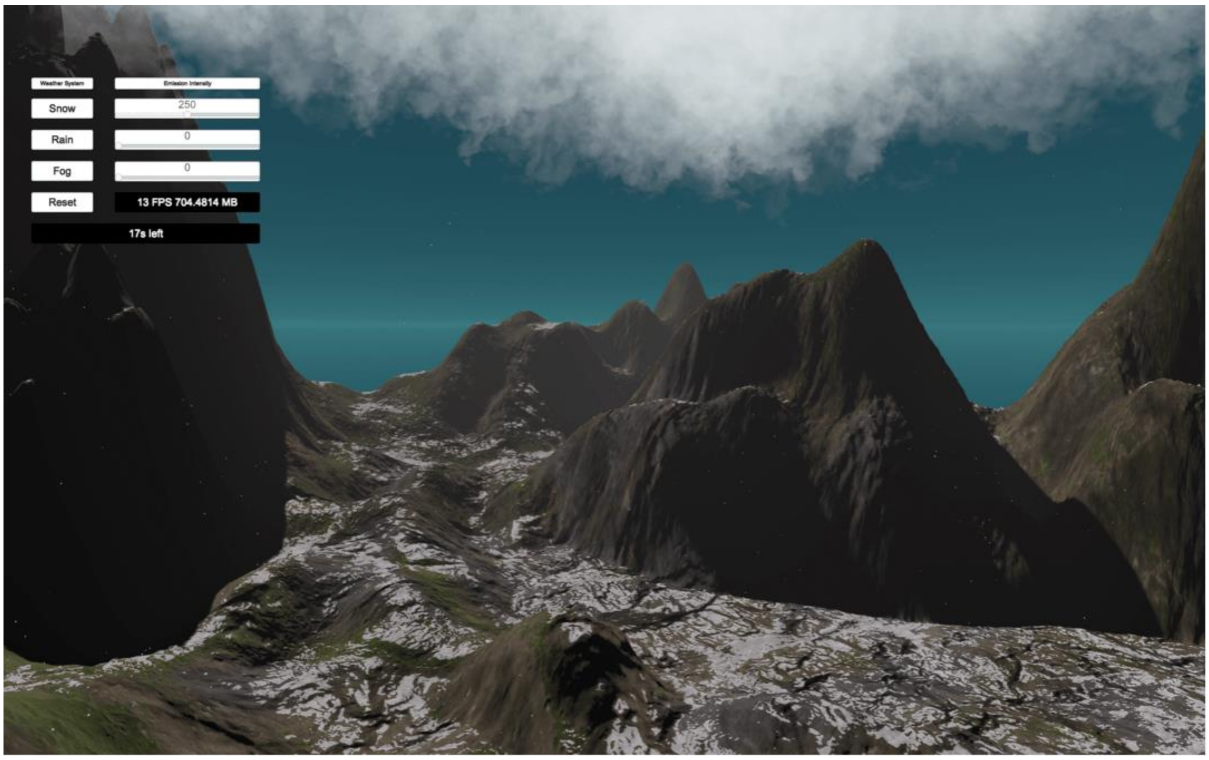


Figure 29 Snow system

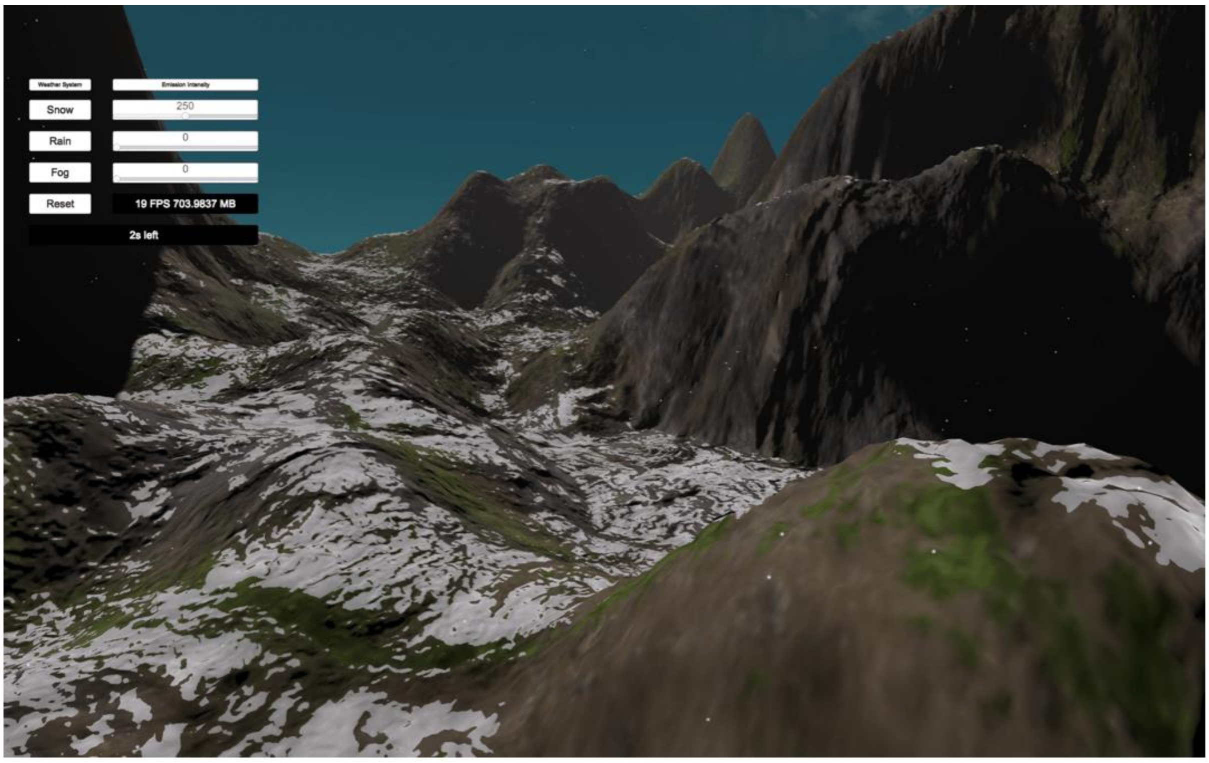


Figure 28 Closer view of the snow build

Figure 30 and 31 show the rain system in use. Note the rain ripples on the ground which correspond to where the rain particles hit the terrain.

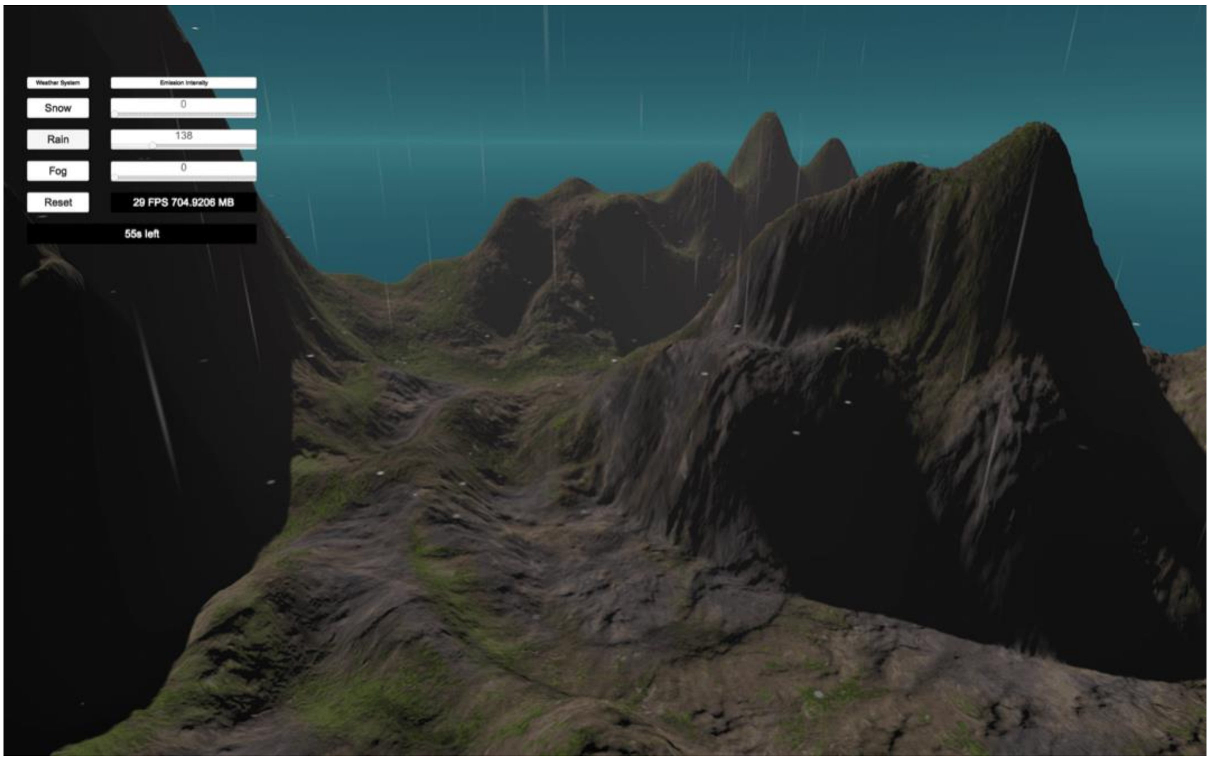


Figure 31 Rain system

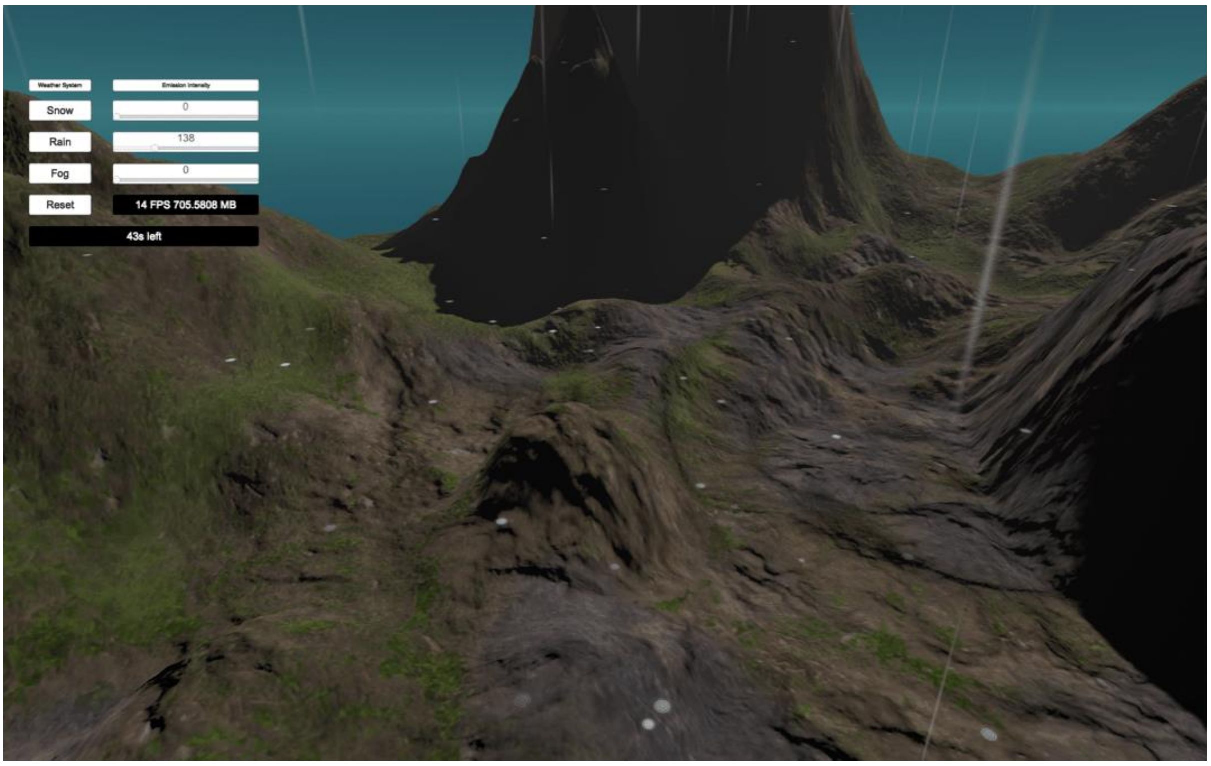
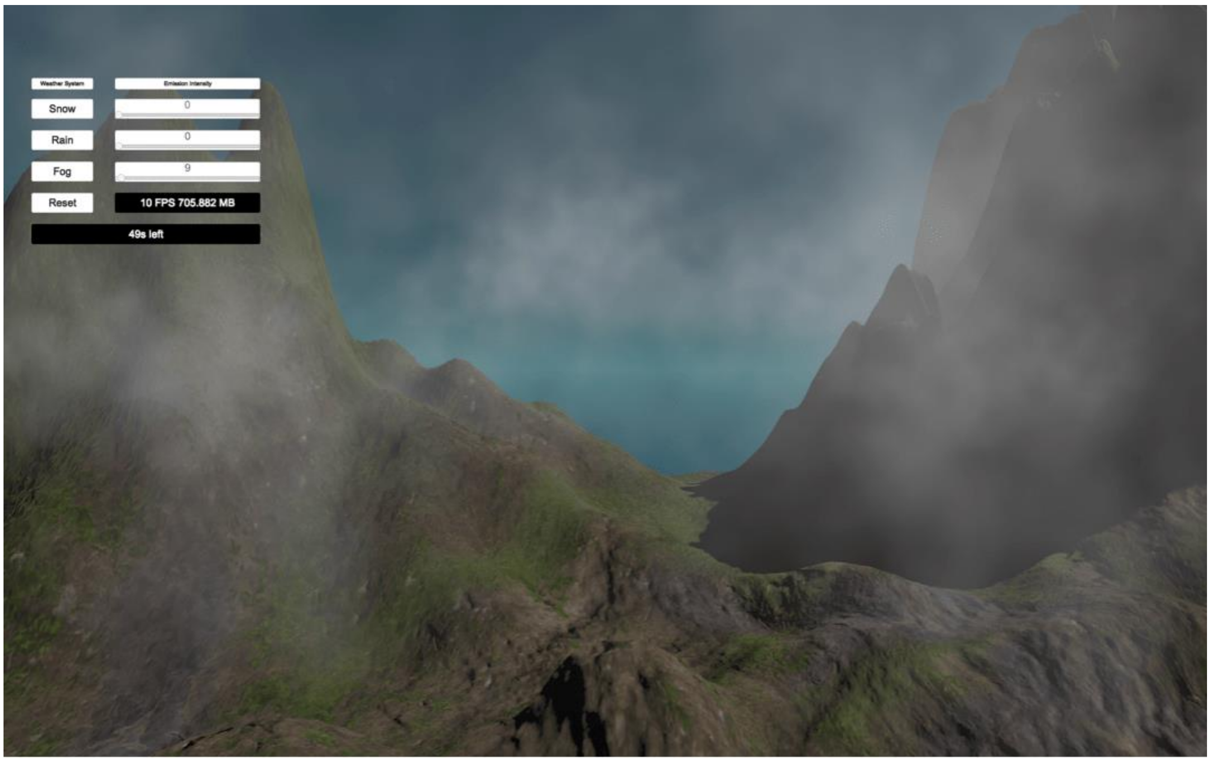


Figure 30 Closer view of the ripples

Finally, figure 32 portrays the fog system. Consider that this is the system set to 9 emission rate.



Weather System	Emission Intensity
Snow	0
Rain	0
Fog	9
Reset	10 FPS 705.882 MB

49s left

Figure 32 Fog system

3.5 Project demonstration links

Below are links to videos that record the project in play.

https://youtu.be/XcdphUBliOk
Alternative link: https://newcastle-my.sharepoint.com/:v:/r/personal/b8011260_newcastle_ac_uk/Documents/Dissertation%20Demo/180112607%20-%20CSC3095%20-%20Graphical%20Rendering%20of%20Dynamic%20Weather%20Systems%20Demo.mp4?csf=1&web=1&e=Zc4y1G

4 Evaluation

This section covers the project's outcomes through repeated testing of the system.

4.1 Methodology

I first exported the project to an executable. I learnt how to do this by following Brackeys tutorial titled "How to BUILD/EXPORT your Game in Unity (Windows | Mac | WebGL)". [44] This showed me the process of exporting the project so it will be running in its own application, not hindered by Unity's background processes.

As previously mentioned in Section 3, the hardware used to develop is the same that will test the project. Other than the weather system program, no other applications were running during testing, and the laptop was plugged into the mains for the whole duration.

In terms of the project, the only consistent game object throughout the weather systems will be the terrain and the clouds. The testing, however, will not account for the player moving around the world using the camera.

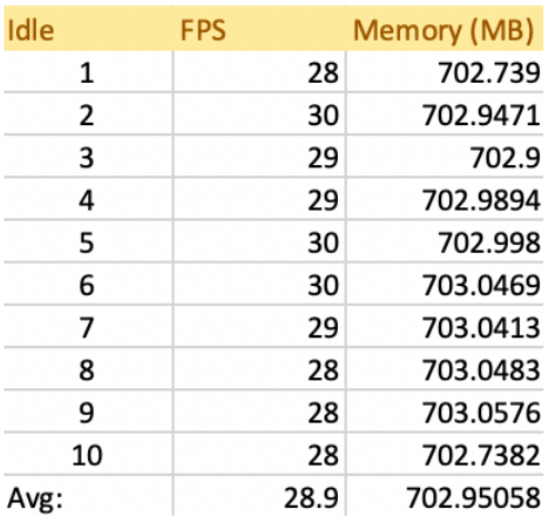
4.2 Test Plan

To test the project, I will bring back the high-level aim: "Investigating the production of realistic weather systems efficiently–exploring the balance between performance and appearance." I will run each weather system 10 times and monitor the average FPS and memory usage over 60 seconds. This will give me a fair average for each system.

On top of this, as each weather condition has different emission options from 0 to 500, I will test the rates at 100, 300 and 500. This also includes the system at idle.

4.3 Machine at idle

To have a baseline to compare to, I ran the program with no systems running on it.



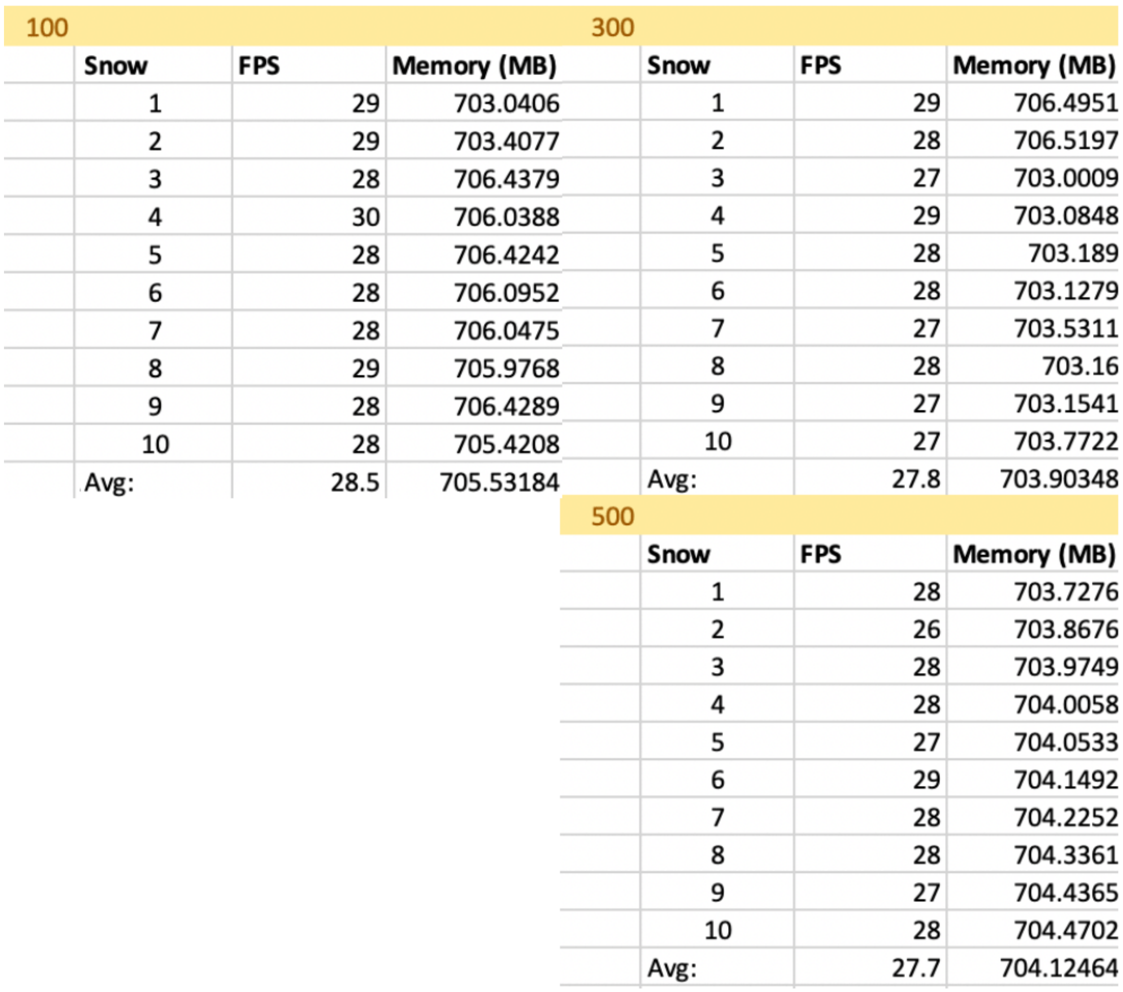
Idle	FPS	Memory (MB)
1	28	702.739
2	30	702.9471
3	29	702.9
4	29	702.9894
5	30	702.998
6	30	703.0469
7	29	703.0413
8	28	703.0483
9	28	703.0576
10	28	702.7382
Avg:	28.9	702.95058

Figure 33 Testing results at idle

Figure 33 shows the results are an average of 28.9 FPS, using 702.95MB (to two decimal places) memory.

To my surprise, even at idle the best performance achievable on my machine was 30 FPS. Prior to this, I was expecting a maximum of 60 FPS, however, this could be due to machine limitations, and I will expand on this further in my evaluation.

4.4 Snow



100			
Snow	FPS	Memory (MB)	
1	29	703.0406	
2	29	703.4077	
3	28	706.4379	
4	30	706.0388	
5	28	706.4242	
6	28	706.0952	
7	28	706.0475	
8	29	705.9768	
9	28	706.4289	
10	28	705.4208	
Avg:	28.5	705.53184	

300		
Snow	FPS	Memory (MB)
1	29	706.4951
2	28	706.5197
3	27	703.0009
4	29	703.0848
5	28	703.189
6	28	703.1279
7	27	703.5311
8	28	703.16
9	27	703.1541
10	27	703.7722
Avg:	27.8	703.90348

500		
Snow	FPS	Memory (MB)
1	28	703.7276
2	26	703.8676
3	28	703.9749
4	28	704.0058
5	27	704.0533
6	29	704.1492
7	28	704.2252
8	28	704.3361
9	27	704.4365
10	28	704.4702
Avg:	27.7	704.12464

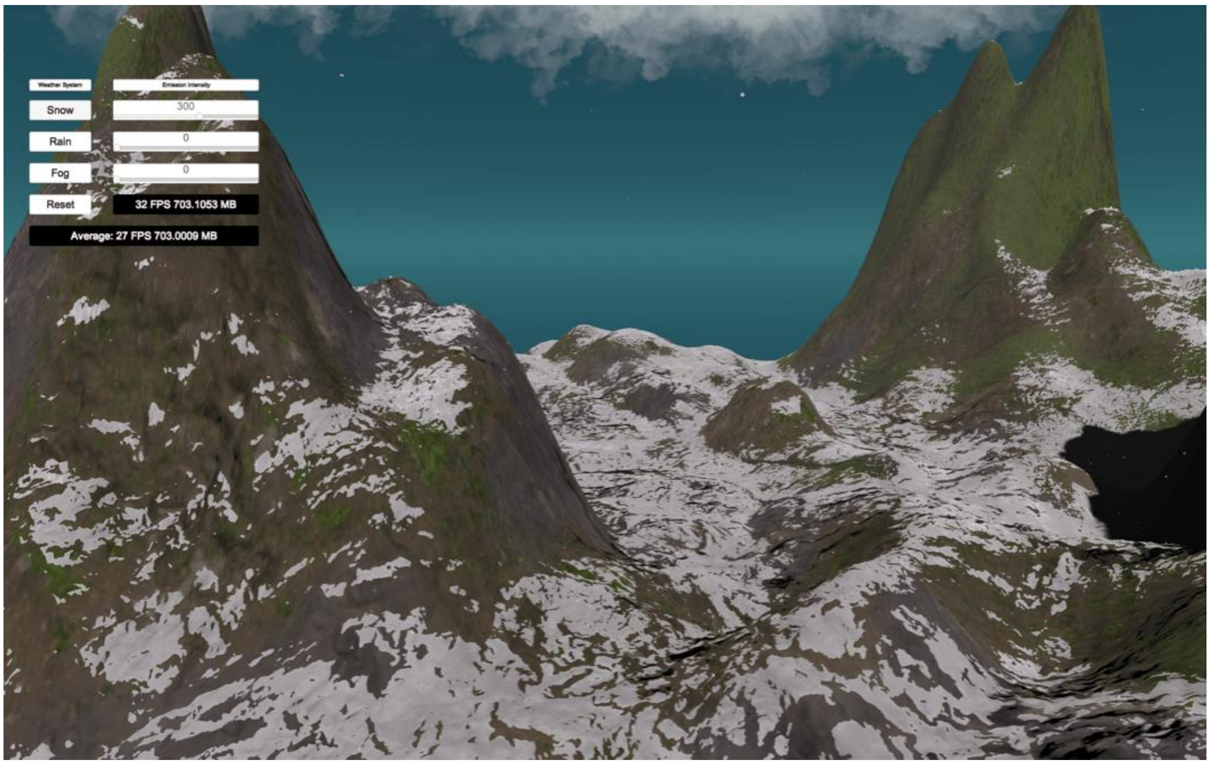
Figure 34 Snow test results

The results are an average of:

Emission	**FPS**	**Memory (MB)**
100	28.5	705.53
300	27.8	703.90
500	27.7	704.12

From Figure 34 above and the supporting table, it is evident that snow is highly optimised. Running the program and starting the snow system was consistently responsive and fast, the snow building on the terrain ran smoothly and there were no issues during runtime.

When compared to the system at idle, there is just over 1 drop in frames, and the memory usage is slightly higher by 2 but other than that the system performs very well.



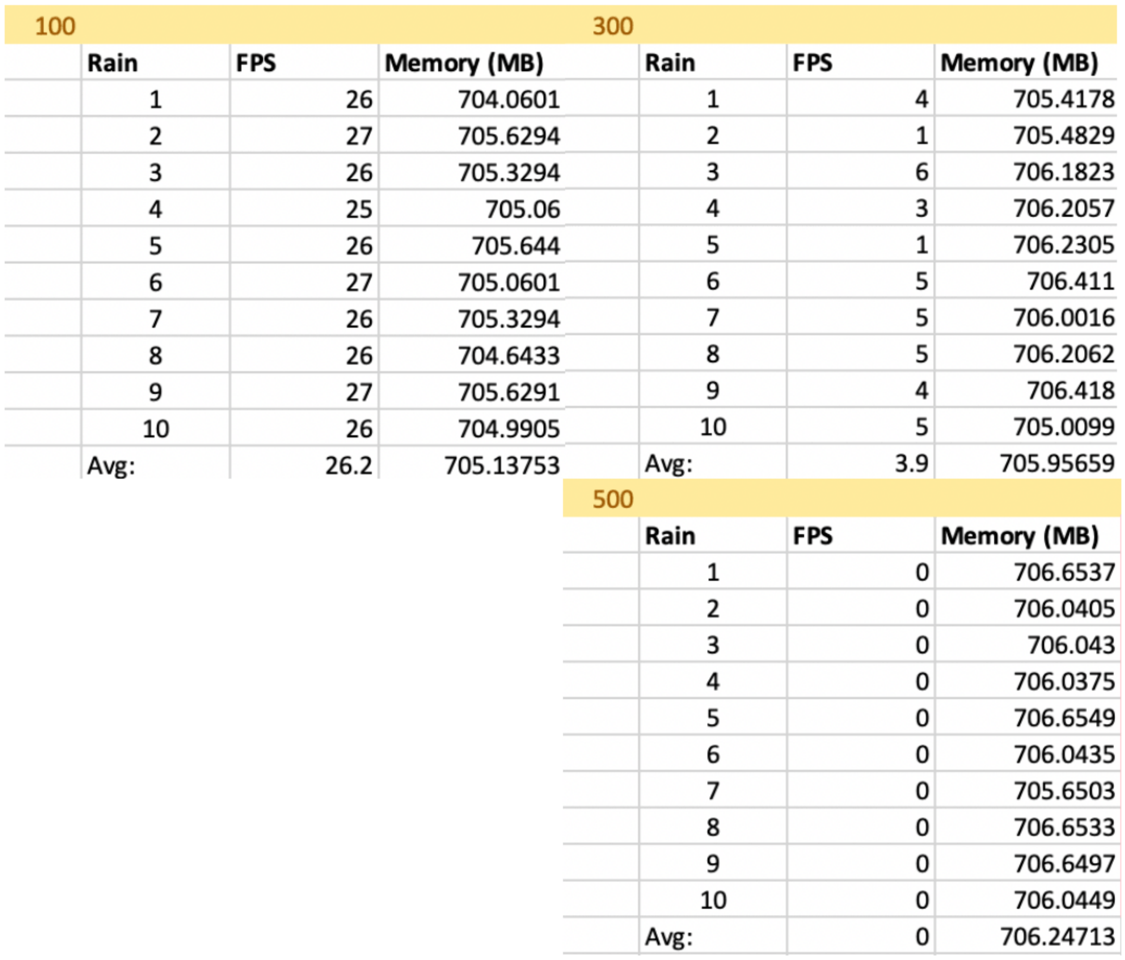
Snow 300
Rain 0
Fog 0

```
32 FPS 703.1053 MB
Average: 27 FPS 703.0009 MB
```

Figure 35 Screenshot of snow at 300 emission

Figure 35 above shows the game world being manipulated by various scripts, yet still maintaining a stead sub-30 FPS.

4.5 Rain



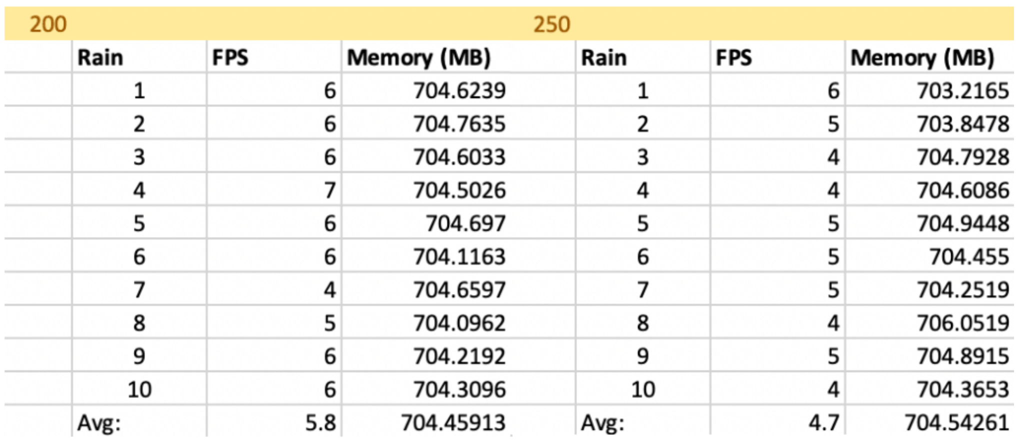
100			300		
Rain	FPS	Memory (MB)	Rain	FPS	Memory (MB)
1	26	704.0601	1	4	705.4178
2	27	705.6294	2	1	705.4829
3	26	705.3294	3	6	706.1823
4	25	705.06	4	3	706.2057
5	26	705.644	5	1	706.2305
6	27	705.0601	6	5	706.411
7	26	705.3294	7	5	706.0016
8	26	704.6433	8	5	706.2062
9	27	705.6291	9	4	706.418
10	26	704.9905	10	5	705.0099
Avg:	26.2	705.13753	Avg:	3.9	705.95659

500		
Rain	FPS	Memory (MB)
1	0	706.6537
2	0	706.0405
3	0	706.043
4	0	706.0375
5	0	706.6549
6	0	706.0435
7	0	705.6503
8	0	706.6533
9	0	706.6497
10	0	706.0449
Avg:	0	706.24713

Figure 36 Rain test results

Figure 36 shows alarming results. Rain does well when set to a value of 100 for emission, but performance drops rapidly when the value is increased—to a final result of 0 FPS when the emission is maxed to 500.

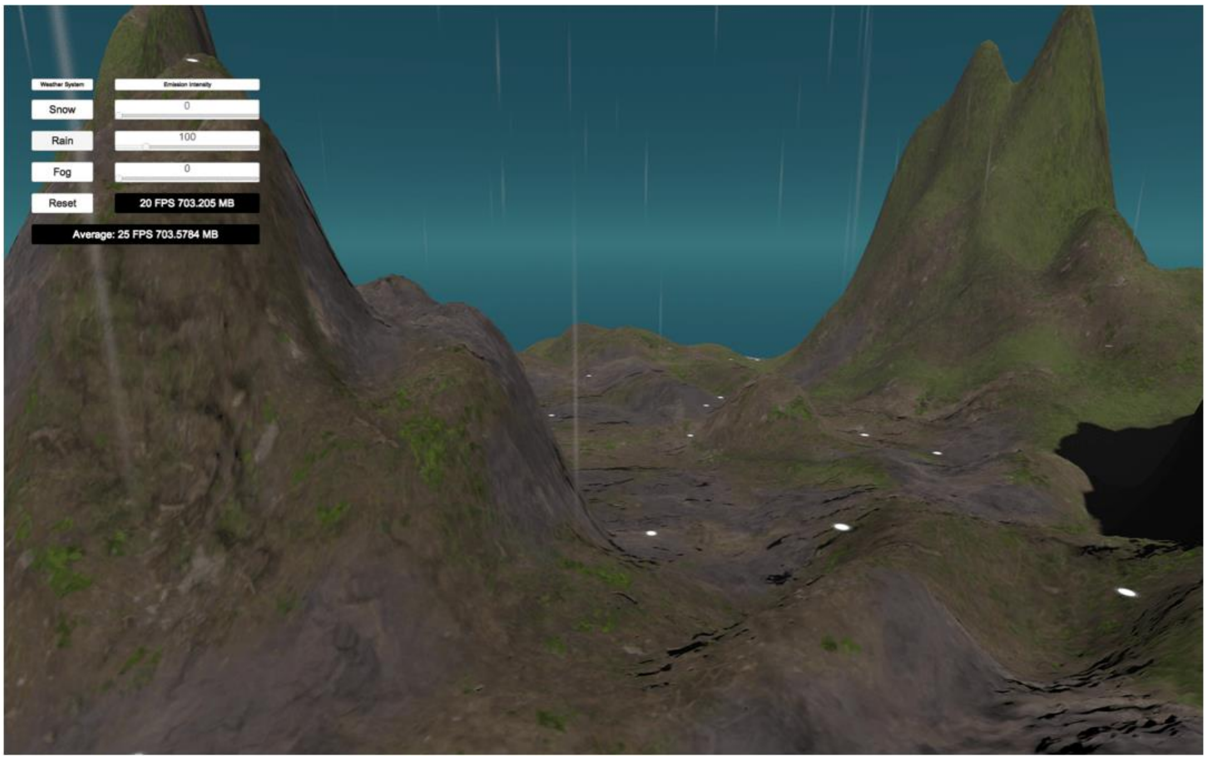
Because of these results, I tested two more emission rates in case it was due to the addition of thunder (as values from 250 will create thunder and its sound).



200			250		
Rain	FPS	Memory (MB)	Rain	FPS	Memory (MB)
1	6	704.6239	1	6	703.2165
2	6	704.7635	2	5	703.8478
3	6	704.6033	3	4	704.7928
4	7	704.5026	4	4	704.6086
5	6	704.697	5	5	704.9448
6	6	704.1163	6	5	704.455
7	4	704.6597	7	5	704.2519
8	5	704.0962	8	4	706.0519
9	6	704.2192	9	5	704.8915
10	6	704.3096	10	4	704.3653
Avg:	5.8	704.45913	Avg:	4.7	704.54261

Figure 37 Rain test results cont.

Figure 37 shows disappointing results. While the frames are slightly better in comparison to 300 and over, in a game setting these results are unplayable.



```
Snow  0
Rain  100
Fog   0
20 FPS 703.205 MB
Average: 25 FPS 703.5784 MB
```

Figure 38 Screenshot of rain at 300 emission

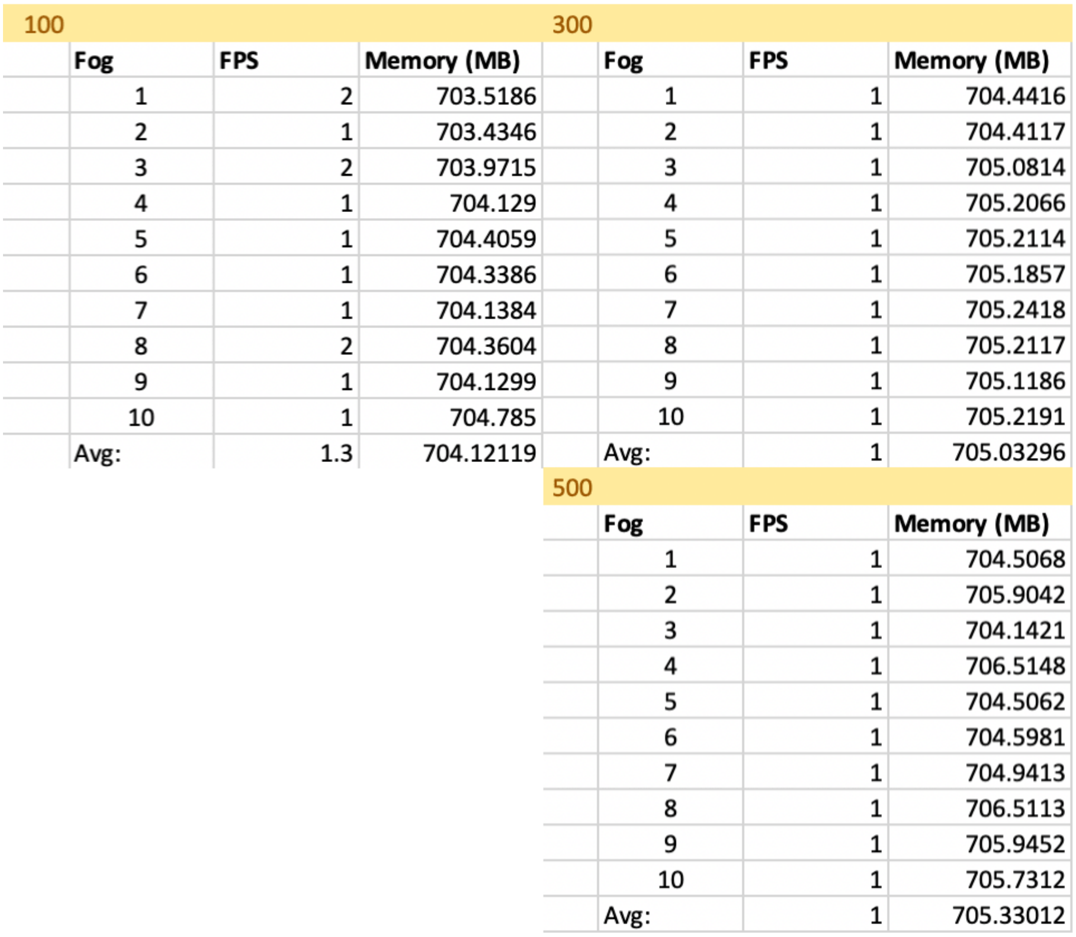
Figure 38 portrays the best emission result for rain, and I feel that the particles are slightly too sparse to accurately represent the weather condition.

The results are an average of:

Emission	**FPS**	**Memory (MB)**
100	26.2	705.53
200	5.8	704.46
250	4.7	704.34
300	3.9	705.96
500	0	706.25

4.6 Fog

Figure 39 also shows poor results but these were to be expected due to fog being unoptimised during development. However, I did not expect the results to be this low. An emission of 100 produced 1.3—its best. Similar to rain, this would certainly be unplayable in a game.



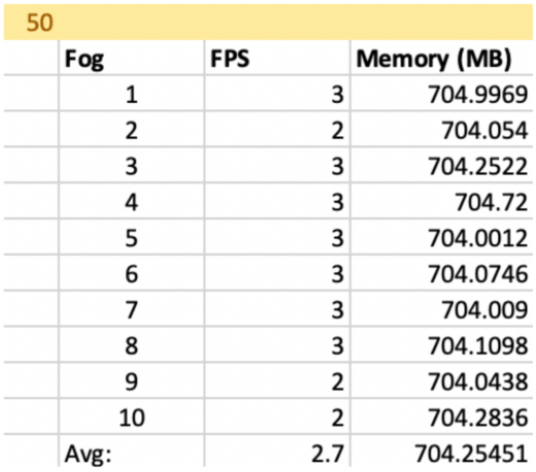
100			
	Fog	FPS	Memory (MB)
	1	2	703.5186
	2	1	703.4346
	3	2	703.9715
	4	1	704.129
	5	1	704.4059
	6	1	704.3386
	7	1	704.1384
	8	2	704.3604
	9	1	704.1299
	10	1	704.785
	Avg:	1.3	704.12119

300			
	Fog	FPS	Memory (MB)
	1	1	704.4416
	2	1	704.4117
	3	1	705.0814
	4	1	705.2066
	5	1	705.2114
	6	1	705.1857
	7	1	705.2418
	8	1	705.2117
	9	1	705.1186
	10	1	705.2191
	Avg:	1	705.03296

500			
	Fog	FPS	Memory (MB)
	1	1	704.5068
	2	1	705.9042
	3	1	704.1421
	4	1	706.5148
	5	1	704.5062
	6	1	704.5981
	7	1	704.9413
	8	1	706.5113
	9	1	705.9452
	10	1	705.7312
	Avg:	1	705.33012

Figure 39 Fog test results

Moreover, I tested fog at 50 emission to see if results would improve, unfortunately, they didn't. As shown in Figure 40 below, the highest average frame was 3.

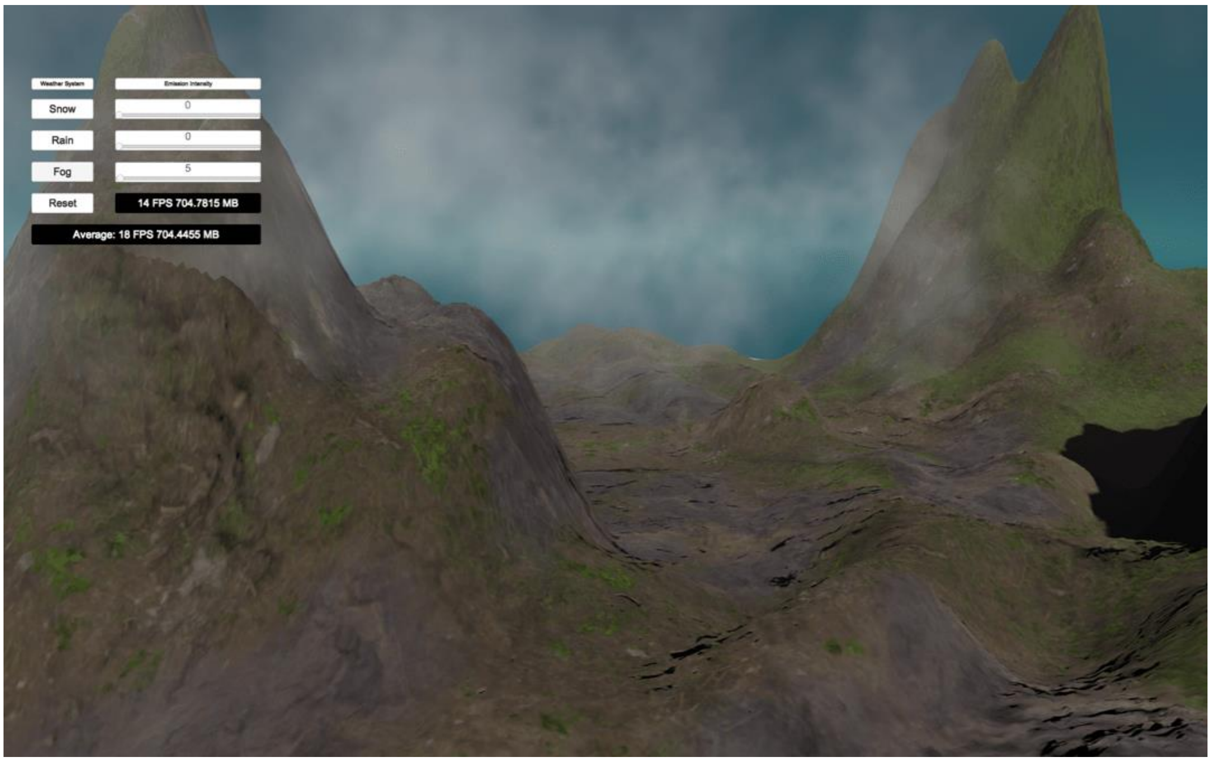


50			
	Fog	FPS	Memory (MB)
	1	3	704.9969
	2	2	704.054
	3	3	704.2522
	4	3	704.72
	5	3	704.0012
	6	3	704.0746
	7	3	704.009
	8	3	704.1098
	9	2	704.0438
	10	2	704.2836
	Avg:	2.7	704.25451

Figure 40 Fog test results cont.

To find the best results for fog I also briefly explored 25, resulting in 7 FPS and 704.2147MB. Then 10, resulting in 13 FPS and 704.2518MB.

Still unsatisfied, I settled on an emission rate of 5, resulting in 18 FPS and 704.4455MB.



Weather System	Emission Intensity
Snow	0
Rain	0
Fog	5

```
14 FPS 704.7815 MB
Average: 18 FPS 704.4455 MB
```

Figure 41 Screenshot of fog at 5 emission

Figure 41 shows the visual outcome of emission at 5. I feel that it does not represent fog well; had the particle system worked flawlessly at 500 emission, it would produce a fuller blanket that obscures the player's camera.

The results are an average of:

Emission	**FPS**	**Memory (MB)**
5	18	704.49
10	13	704.25
25	7	704.21
50	2.7	704.25
100	1.3	704.12
300	1	705.22
500	1	705.33

4.7 Summary

Throughout this section, I tested each system at 100, 300 and 500 emission (plus additional independent emission testing) and recorded their FPS value and RAM usage, and the actual visual output of the systems. It is vital to note that while performance plays a big role in game design, the average player will not be overjoyed about the low memory overhead if the visuals are poor.

To begin with, we will look at fog. The results were shocking, to say the least, with only 5 particles emitted per second, it is hardly a weather condition in practice. As mentioned during development, it was already running at low frames even when in the Unity editor.

The memory needed to produce even the lowest emission rates is also too high for the graphical output. To conclude, the results show that this method of producing fog where I import a custom material and then produce it on the screen using particle systems is a bad choice.

Next is rain. While the performance results are good, with a solid 22 FPS (if the baseline was doubled to be 60 FPS, this would be 44 FPS) which is a decent outcome, but the graphical result is sub-par. During development in the Unity editor, running the rain at max emission was smooth and there were no hiccups in frames. However, running inside the standalone build was a different story.

Rain also produced the highest RAM overhead compared to the other two systems. This must be because of the 4 particle systems taking place at the max setting: rain, ripples, splashes and lightning. On top of the two overlapping sounds of rain and thunder.

On the topic of sounds, the output was smooth and there were no issues. The reasoning behind the great drop in performance must be because of the two sound files playing on top of each other, which requires an inordinate amount of power. Furthermore, both files could have been smaller sound bites.

Moreover, during testing, the frames were above 25 until 10 seconds in when the FPS started dropping. This shows that it must be the ripples and splashes that occur after the rain collides with the terrain because these happen and persist after a few seconds.

Snow proves to be the frontrunner when considering frames and memory usage since the average FPS is close to the system at idle, and that is most desirable. As for RAM, the tests generally show a gradual increase in MB used in relation to the emission rate. At its maximum emission, the average memory needed to power the project was only 704.12MB which is only slightly larger than idle—a great result. This shows the scripts were successful and their compatibility with the materials is great.

When comparing the snow to rain, I feel that the performance difference could also be due to snow only using the default settings on the particles themselves. For example, the default particle had to be altered for rain (elongating and subsequently making each drop larger in comparison), while the snow was just a manipulation of the way the particles were released.

Additionally, it is important to note that snow also had its own script due to unrelated reasons to frames and memory. While previous research into the effects of combining all of the code into one file showed to have no effect, the MasterScript did handle fog and rain at the same time.

A subsequent investigation will have to be done to see which of the above factors affected performance the most.

5 Conclusion

5.1 Satisfaction of aims and objectives

The original aim was to investigate the provision of efficient and realistic looking weather effects, with regards to performance against appearance. This has been broken down into manageable objectives as detailed, followed by an assessment on its fulfilment:

Objective 1: *To engage in background reading on rendering weather, and investigate the current market to identify how games use weather systems effectively.*

For the background research, I looked into rendering in general and at least one paper on each weather system, with an eye for different rendering methods. For example, Guo et al’s research into heterogeneous fog led me to implement fog with four different materials, ensuring diversity.

Objective 2: *Research and identify key weather systems to implement, with the thought of producing a well-rounded project in mind.*

My reading helped me in choosing systems, and I settled on snow, rain and fog. I selected these because the snow would require additional regard to displaying changes on the terrain; rain allows for further expansion into storms and raindrop physics, and fog to test the effects of a large particle system over the whole environment.

Objective 3: *Design an appropriate game world to apply the systems to.*

I successfully created a custom terrain with dips and hills to allow the snow build script to shine.

Objective 4: *Implement snowfall and its accompanying conditions.*

Snow is well implemented with its own particle system and the snow build script modifying the environment with a snow blanket.

Objective 5: *Implement rainfall and its accompanying conditions.*

Rain is implemented alongside splashes and ripples on the terrain. On top of this, lightning is present once emission increases enough.

Objective 6: *Implement fog and its accompanying conditions.*

Fog is also implemented and creates a large cover over between the dips and hills of the ground.

Objective 7: *Test the program repeatedly and ensure each weather system works independently and properly.*

An appropriate test plan, demonstrating average FPS and memory usage values instead of one-take performance numbers, was devised and followed.

Objective 8: *Evaluate if the program runs smoothly in full screen with a clear written assessment.*

An analysis into why these testing outcomes were achieved and a visual representation of the best conditions of each system.

5.2 Personal development

Throughout this project, I have achieved many personal goals. Prior to undertaking this dissertation, I had limited knowledge of C# and the Unity platform, however, with dissertations' tendency to be individually led, I needed to depend on myself for learning how to navigate a new programming language and development platform. Luckily, Unity is a well-documented program with a multitude of tutorials and help online so encountering difficult problems were easily avoided. My previous experience in other programming languages meant I developed transferrable skills with C# despite not being too well versed in it.

Time management wise, I abided by my proposed schedule. Development was done without rushing, and it allowed for ample time for testing despite each system being tested multiple times.

5.3 Challenges

5.3.1 Technology restrictions

Despite the COVID-19 pandemic restricting my access to dedicated University computers within the Urban Sciences Building, I managed well with my MacBook. I found that my biggest concern was the restriction of software on a non-Windows operating system, but the technologies I chose to realise this project were all accessible on macOS.

In hindsight, I think I would have developed the project on my own personal computer anyway. This is due to my familiarity with the operating system and the ease of use, regardless of building closures.

As previously discussed during the evaluation, the highest FPS achieved after building the project was 30 FPS which dismantles the high-level aim of developing with 60 FPS in mind. While 60 is the standard nowadays, 30 is also an acceptable baseline for gaming from my experience, as long as it is kept at a stable framerate.

5.3.2 Rain clouds disappearing

During the final weeks of development, the rain system's clouds stopped appearing on screen. I attempted many ways of fixing the issue but to no avail. The same snippet of code for the clouds can be copied and pasted from snow onto rain, with the appropriate particle systems turned on/off, and it would still decline to show. Furthermore, when testing to see if it was the cloud material that was the problem, the same material would show for snow with no issue. All of these attempts had every other part of the rain system working, including the sound, splashes and ripples.

Because of the time this issue arose in terms of the academic year, there was no time to fix the bug due to the nearing deadline of the project. I also consulted my supervisors and they agreed that I leave the bug as it is and focus on finishing the dissertation write up.

With more time, I could have fixed this issue through more rigorous testing and hardware changes. Although currently, I feel it could be due to the number of particle systems running at any one time.

5.3.3 Unity editor performance

As briefly mentioned in Section 4.7, a factor I did not consider during development is that running the program within the Unity editor results in differing outcomes in comparison to a separate executable file. Results such as FPS and memory usage were the primary benchmarks I used to calibrate my particle systems during development, adjusting them according to the results of what was presented.

While the natural decision moving forward is to build and export the program after every change, it is extremely inefficient and time-consuming. In the future, I would invest more time into the development section, export the game after each system is made in order to record the performance, and then adjust the parameters. This would present the program to the developer with more accurate performance outcomes.

5.4 Future development

5.4.1 Script organisation

In Section 3.3.10 I explained my reasoning in combining most of the particle system scripts into one cohesive file to slightly improve performance. In practice, I found that this could have been the reason why rain and fog performed so badly in comparison to snow, which had its own script. With more time, it would be important to look into if this design choice resulted in the poor results.

Though the difference it made was for me as a developer was great in efficiency, in future developments of this project I would have to split the files again. Not only for organisational purposes that will be required in a bigger team, but for the intent of allowing multiple developers to work on the same project but different sections.

5.4.2 Glossy terrain

Rain should cause the environment to become glossier and create puddles to simulate being slick with rain. Raindrops should also affect the parts of the floor with water. With more time I believe this will increase immersion, alongside better hardware that allows for running with greater than 10 FPS.

5.4.3 Vertex height adjustment

Right now, snow only “builds up” on the terrain in the sense of drawing white over the environment in relation to each vertices position against the top of the game world. A better way of implementation would be to increase the height of the whitened parts of the terrain to look more natural.

Furthermore, with puddles being created with rain, the height of the terrain must increase as well. With raindrops causing small indents in the puddles in the environment.

5.4.4 Wind zones

Prior to the development of the project, I initially planned to implement Unity’s wind zones game object. These are areas that move the particles in a certain direction, imitating wind. This would have been applied to the snow system, in the same manner that emission of over 250 in rain causes lightning to appear, wind zones would hypothetically appear after 250 too. In my opinion, this is a vital feature of blizzards.

However, because of the time constraint, I had to cut this feature out of the program. With more time, I would have liked to explore this section further.

5.4.5 Alternative rain rendering technique

A tutorial by mrwasd titled "Unity3D Rain Tutorial" [45] demonstrates a different method of rendering rain. Instead of using the default Unity particle system material which is a simple white image, they used a custom material seen in Figure 43 below.

This method is similar to the one I employed for clouds and fog. The custom material is created on a black background, so Unity can render the lighter areas of the image in the scene, reproducing rain.

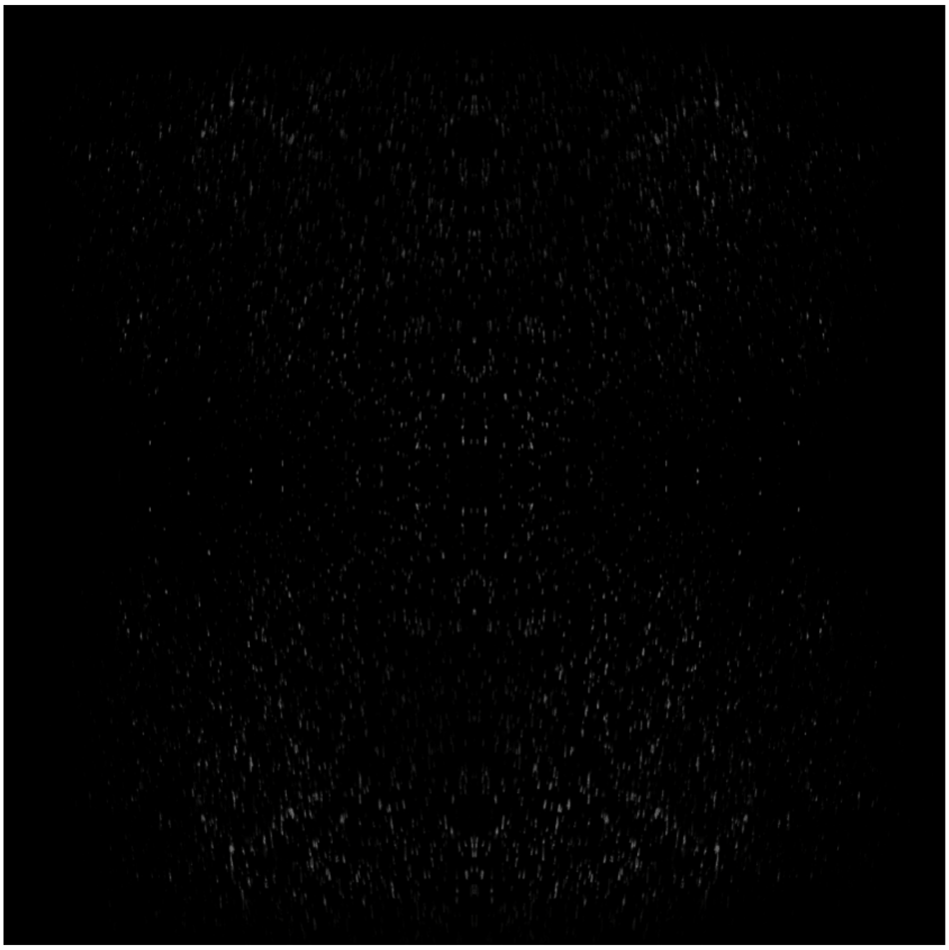


Figure 42 Custom rain sheet that mrwasd used in their YouTube tutorial

This method, however, makes use of a (now) legacy shader. This means that the shader is no longer supported or being updated with newer releases because its functionality has been removed or deprecated. [46] For the method to be applicable for future development, it must be employable with the current shaders in the latest Unity release.

I attempted this technique briefly and found that my MacBook was unresponsive during the rendering and calibration process of the particle system. The image file was too large and I was then unable to proceed further.

Regardless, this is an insight into other rendering techniques available when investigating this project further, and with a more powerful machine.

6 Bibliography

[1] The Game Awards, “2017 | History | The Game Awards,” 2017. [Online]. Available: https://thegameawards.com/history/year-2017.

[2] Intellivision, “Intellivision Legacy,” [Online]. Available: https://intellivision.com/legacy. [Accessed 8 April 2021].

[3] B. Edwards, “Gaming in the Rain: Dramatic Weather in Classic Video Games,” PCMag, 24 August 2019. [Online]. Available: https://uk.pcmag.com/gallery/122273/gaming-in-the-rain-dramatic-weather-in-classic-video-games?p=1. [Accessed 8 May 2021].

[4] H. Moravec, “When will computer hardware match the human brain?,” *Journal of Evolution and Technology,* vol. 1, 1998.

[5] P. Rousseau, V. Jolivet and D. Ghazanfarpour, “Realistic Real-Time Rain Rendering,” *Computers & Graphics,* vol. 30, no. 4, pp. 507-518, 2006.

[6] M. Langer, L. Zhang, A. Klein, A. Bhatia, J. Pereira and D. Rekhi, “A Spectral-Particle Hybrid Method for Rendering Falling Snow,” *Eurographics Symposium on Rendering,* pp. 217-226, 2004.

[7] P. Fearing, “Computer Modelling of Fallen Snow,” *SIGGRAPH '00: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques,* pp. 37-46, 2000.

[8] F. Guo, J. Tang and X. Xiao, “Foggy Scene Rendering Based on Transmission Map Estimation,” *International Journal of Computer Games Technology,* pp. 1-13, 2014.

[9] F. Häggström, “Real-Time Rendering of Volumetric Clouds,” *Master of Science in Engineering: Umeå University,* 2018.

[10] E. Ferrara, S. Catanese, G. Fiumara and F. Pagano, “Rendering of 3D Dynamic Virtual Environments,” *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques,* 2011.

[11] WolframMathWorld, “Normal Vector,” 2021. [Online]. Available: https://mathworld.wolfram.com/NormalVector.html. [Accessed 3 May 2021].

[12] Unity, “MonoBehaviour,” 25 April 2021. [Online]. Available: https://docs.unity3d.com/ScriptReference/MonoBehaviour.html. [Accessed 3 May 2021].

[13] User Benchmark: GPU, “Nvidia GTX 1050 vs Intel Iris Plus 650,” [Online]. Available: https://gpu.userbenchmark.com/Compare/Nvidia-GTX-1050-vs-Intel--Iris-Plus-650-Mobile-Kaby-Lake/3650vsm367939. [Accessed 8 April 2021].

[14] Transform Interactive, “The 5 Best Uses for Unity (Besides Game Development),” [Online]. Available: https://transforminteractive.com/5-best-uses-for-unity/. [Accessed 8 April 2021].

[15] Unity, [Online]. Available: https://unity.com/. [Accessed 8 April 2021].

[16] R. Dillet, “Unity CEO says half of all games are built on Unity,” Tech Crunch, 5 September 2018. [Online]. Available: https://techcrunch.com/2018/09/05/unity-ceo-says-half-of-all-games-are-built-on-unity/. [Accessed 8 April 2021].

[17] Unity, “High Definition Render Pipeline overview,” 2020. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@11.0/manual/index.html. [Accessed 6 November 2020].

[18] Unity, “Materials, Shaders & Textures,” 12 July 2017. [Online]. Available: https://docs.unity3d.com/560/Documentation/Manual/Shaders.html#:~:text=Rendering%20in%20Unity%20is%20done%20with%20Materials%2C%20Shaders%20and%20Textures.&text=Shaders%20are%20small%20scripts%20that,input%20and%20the%20Material%20configuration.. [Accessed 11 April 2021].

[19] OpenGL Wiki, “Cg,” 1 February 2021. [Online]. Available: https://www.khronos.org/opengl/wiki/Cg. [Accessed 11 April 2021].

[20] Unity, “Introduction to Particle Systems,” [Online]. Available: https://learn.unity.com/tutorial/introduction-to-particle-systems#6025fdd9edbc2a112d4f0133. [Accessed 5 November 2020].

[21] G. A. Prod., “Unity 2017 - Game VFX - Realistic Rain Effect,” 23 December 2017. [Online]. Available: https://www.youtube.com/watch?v=Ph3FvxJJ8AA. [Accessed 6 November 2020].

[22] Mirza, “Unity VFX - Realistic Snow (Particle System Tutorial),” 24 November 2016. [Online]. Available: https://www.youtube.com/watch?v=b8oVAS9IdZM. [Accessed 2 November 2020].

[23] Brackeys, “SNOW in Unity - SHADER GRAPH,” 26 May 2019. [Online]. Available: https://www.youtube.com/watch?v=IC9g5hlfV6o. [Accessed 6 November 2020].

[24] k0rveen, “LowPoly Environment Pack,” 6 October 2017. [Online]. Available: https://assetstore.unity.com/packages/3d/environments/landscapes/lowpoly-environment-pack-99479. [Accessed 10 November 2020].

[25] Unity Forum, “Materials are pink,” 22 November 2018. [Online]. Available: https://forum.unity.com/threads/materials-are-pink.587071/. [Accessed 8 May 2021].

[26] W. o. Zero, “Writing a Snow Covered Shader,” 4 October 2019. [Online]. Available: https://www.youtube.com/watch?v=bY7r6blL1K8. [Accessed 15 November 2020].

[27] b3agz, “LET IT SNOW - Super Simple Snow System for Unity,” 24 December 2019. [Online]. Available: https://www.youtube.com/watch?v=x8nPDrzZ5p8. [Accessed 15 November 2020].

[28] U. Documentation, “WaitForSeconds,” Unity, 2020. [Online]. Available: https://docs.unity3d.com/ScriptReference/WaitForSeconds.html. [Accessed 19 November 2020].

[29] Brackeys, “PROCEDURAL TERRAIN in Unity! - Mesh Generation,” 4 November 2018. [Online]. Available: https://www.youtube.com/watch?v=64NblGkAabk. [Accessed 12 November 2020].

[30] Unity Asset Store, “Mountain Terrain + Rock + Tree,” 31 August 2017. [Online]. Available: https://assetstore.unity.com/packages/3d/environments/landscapes/mountain-terrain-rock-tree-97905. [Accessed 10 November 2020].

[31] Brackeys, “How to make Terrain in Unity!,” YouTube, 23 June 2019. [Online]. Available: https://www.youtube.com/watch?v=MWQv2Bagwgk. [Accessed 7 February 2021].

[32] M. Vasileva, “24 Clouds,” Brusheezy.com, [Online]. Available: https://www.brusheezy.com/brushes/2187-24-clouds. [Accessed 16 November 2020].

[33] TheSenpaiCode, “Unity3d, Different Ways to Make Fog With Lighting & particle Systems,” YouTube, 23 April 2018. [Online]. Available: https://www.youtube.com/watch?v=POFKoQzl5pU. [Accessed 29 February 2021].

[34] I. Vitkovskiy, “Free Fog Photoshop Brushes,” Brusheezy, [Online]. Available: https://www.brusheezy.com/brushes/57325-free-fog-photoshop-brushes. [Accessed 20 February 2021].

[35] Madhavan, “Designing a Button,” Prototypr.io, 9 August 2018. [Online]. Available: https://blog.prototypr.io/designing-a-button-a2ad6c0ec310. [Accessed 15 April 2021].

[36] __MaX__, “Multiple scripts VS Single script (performance, purpose...),” Unity Forum, 5 January 2013. [Online]. Available: https://forum.unity.com/threads/multiple-scripts-vs-single-script-performance-purpose.164942/. [Accessed 15 April 2021].

[37] M. -. S. M. &. Mindfulness, “Nature Sounds: Rain Sounds One Hour for Sleeping, Sleep Aid for Everybody,” YouTube, 09 April 2013. [Online]. Available: https://www.youtube.com/watch?v=Mr9T-943BnE. [Accessed 25 January 2021].

[38] SPANAC, “Thunder Sound No Rain,” Free Sounds Library, 26 March 2019. [Online]. Available: https://www.freesoundslibrary.com/thunder-sound-no-rain/. [Accessed 25 January 2021].

[39] SPANAC, “Wind Blowing Sound Effect,” Free Sounds Library, 05 November 2017. [Online]. Available: https://www.freesoundslibrary.com/wind-blowing-sound-effect/. [Accessed 25 January 2021].

[40] Megan, “How do I use an Audio Source in a script?,” Unity, 2017. [Online]. Available: https://support.unity.com/hc/en-us/articles/206116056-How-do-I-use-an-Audio-Source-in-a-script-. [Accessed 25 January 2021].

[41] J. French, “How to fade audio in Unity: I tested every method, this one's the best,” gamedevbeginner, 29 August 2019. [Online]. Available: https://gamedevbeginner.com/how-to-fade-audio-in-unity-i-tested-every-method-this-ones-the-best/. [Accessed 10 February 2021].

[42] K. Mouse, “Customizable skybox,” Unity Asset Store, 13 July 2020. [Online]. Available: https://assetstore.unity.com/packages/2d/textures-materials/sky/customizable-skybox-174576. [Accessed 26 February 2021].

[43] Unity, “Memory Profiler Module,” 19 April 2021. [Online]. Available: https://docs.unity3d.com/Manual/ProfilerMemory.html. [Accessed 23 April 2021].

[44] Brackeys, “How to BUILD / EXPORT your Game in Unity (Windows | Mac | WebGL),” YouTube, 12 April 2017. [Online]. Available: https://www.youtube.com/watch?v=7nxKAtxGSn8. [Accessed 21 April 2021].

[45] mrwasd, “Unity3D Rain Tutorial,” YouTube, 19 May 2016. [Online]. Available: https://www.youtube.com/watch?v=Ax3WNI-3C60. [Accessed 9 April 2021].

[46] Unity Documentation, “Legacy Shaders,” 19 April 2021. [Online]. Available: https://docs.unity3d.com/Manual/Built-inShaderGuide.html. [Accessed 22 April 2021].

[47] Giant Bomb, “What is regarded to be the first Sim and Strategy game to ever come to a console platform, Utopia sets up two rival nations and hands you the reigns of leader.,” [Online]. Available: https://www.giantbomb.com/utopia/3030-1643/. [Accessed 8 April 2021].

[48] G. A. Prod., “Unity 2017 - Game VFX - Realistic Rain Effect,” 23 December 2017. [Online]. Available: https://www.youtube.com/watch?v=Ph3FvxJJ8AA. [Accessed 17 November 2020].

[49] Apple, “Activity Monitor User Guide,” 2021. [Online]. Available: https://support.apple.com/en-gb/guide/activity-monitor/welcome/mac. [Accessed 22 April 2021].