Creation of a graphical dynamic weather system and an analysis of its performance of the Graphics Processing Unit

By Lucas Chetcuti, 180275902

Supervisor: Dr Graham Morgan

BSc Computer Science
Newcastle University

Word Count: 11450

Abstract

As video games have advanced and become more varied and complex, new systems have been developed which help to improve the immersion and enjoyment players experience when playing modern titles. One of the more common, yet more complex types of these systems are in-game weather systems. Weather in games is not a recent development but with the advancement of hardware, examples which now involve complex particle simulations and physics-based interactions with other assets in the game environment.

This paper aims to see how a weather system can be created with a series of complex behaviours while still being able to use only a small allocation of the underlying hardware's processing capacity. With the performance analysis, this dissertation can be a useful resource for other graphics and game developers on the best practices they should use when designing their own dynamic visual effects and how the mistakes that were made in the creation process could be avoided in the future.

Declaration

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

Acknowledgements

I would like to thank my supervisors Graham Morgan, Gary Ushaw and Richard Davison for helping me throughout the development and writing of this dissertation over the past eight months. Your instructions and advice played a large part in the outcome of this project. I would also like to thank my family who managed to support me during this challenging yet rewarding time and my friends, who were always there to help me when I needed it most.

Table of Contents

1. Introduction

This section provides the overall motivation, scope of the project, and an outline for the aims and objectives of the project which will be used to measure the success of the project. It also gives summaries for every other section of the paper.

1.1 Motivation

Weather systems help to create an added sense of realism and immersion into a video game world. Today, many games ranging from the largest triple-A games down to independent titles now feature some form of in-game weather to help add artistic styling or to work with other in-game systems to help create dynamic gameplay scenarios.

Modern video games are developed with many visual effects to help attain a higher level of graphical fidelity. While this is always done to improve the overall quality of the game, there are many cases where the number of features becomes too costly to the hardware's resources (Barton, 2021) and will start to affect the game performance. This impact can range from a momentary drop-in frame rate to the potential of the game crashing due to the hardware's inability to run the software.

The hope for this project is to be a helpful study on how to make certain features run more efficiently and when it is worth considering the removal of certain features that prove to be too intensive. While it would not be too challenging to create a highly detailed weather system with full access to a computer's processing capacity, this would not be useful for a full game as the weather system would be given a comparatively low allocation of system resources.

1.2 Scope

Weather systems have several aspects that all play a significant role in helping them be a part of a game's environment that elevates the overall aesthetic, gameplay and immersion of the game. The overall goal of this project is to focus on the creation of weather effects that accurately represent the behaviour of different types of weather in the real world while trying to strike a balance and sacrifice less significant aspects to improve the overall performance of the weather system at runtime and reduce the overall cost on the underlying hardware. For this reason, the project does not look at creating a visually appealing system. Where visual assets such as textures and 3D meshes are used, this is purely to help visualise certain aspects of the weather system more clearly but there will be little thought given to the overall artistic appeal of the system in the project as a whole.

1.3 Aims and Objectives

The aim of the project is:

"Developing a weather system within the Unity Game's Engine that does require an extensive amount of the underlying hardware's processing power."

To achieve this aim, these are the 5 main objectives for the project:

1) Study how industry developers implement weather systems and other particle effects into games available in the market.
2) Develop an understanding of the Unity Game Engine.
3) Create a visual effects system that can simulate multiple types of weather effects.
4) Design the effects to have interactive elements with other in-game elements.
5) Optimise the project so it does not become too costly on processing power.

1.4 Paper Outline

Research and Background

Outline of research in how to create an effective weather system within a games engine and further reading from professionals on how particle systems are created within the industry.

Methodology

The approach to how the project will be developed using the research from the prior section and what software, tools and techniques will be used to create the final version of the weather system.

Results

This section will be a log of the framerate and GPU usage of the project while it is running. This will be tested through the Unity Profiler and the results will be used to evaluate the performance of the Weather System in the next section.

Evaluation

The evaluation will be used to discuss the results that were gathered in the previous section. This will highlight what areas of the project may have had the greatest impact on performance and to highlight issues in the result gathering process.

Conclusion

This will be a summary of the final thoughts of the entire project and will largely focus on the areas of:

- The overall outcome of the project and how it satisfied the aims and objectives.
- My own personal development throughout this project.
- How the project could be improved going forward.

2. Background

The background will look at the previous work of graphical weather systems, focusing on how they are utilised in media and the underlying technologies which are used to create them.

2.1 History of weather systems used in the games industry.

In the early 90s, it was only higher budget 2D games such as Super Metroid (Nintendo, 1994) and A Link to The Past (Nintendo, 1991) that featured weather effects. These were mainly done using animated backgrounds and foreground sprite effects. These effects were not seen commonly in this era as the memory on contemporary hardware was limited, the Super Nintendo Entertainment System only had 16 bits of CPU memory and 1MB of RAM which it could utilize. As a result, aesthetic features would have been sacrificed for more vital aspects of a game such as level design and music. As discussed by Jonasson and Purho (Jonasson and Purho, 2012), for games where developers were able to include weather effects, they helped to give added 'juice' to their games and mainly have the intended effect of influencing the users' immersion within the game.

Figure 1: The rain effect demonstrated in A Link to the Past (left) and Super Metroid (right). These effects are only used at key story points in both games, most likely because they would be too costly on the Super Nintendo.

Later in the decade newer hardware like the PlayStation and Nintendo 64 had double and quadruple the CPU capacity of previous hardware. This generation of hardware was where we see the earliest example of GPUs which made the task of rendering 3D graphics much simpler (The History of Gaming: The evolution of GPUs, 2020). This is what lead to the larger shift from 2D to 3D games by much of the video games industry and allowed for a greater use of weather systems in games. While these were mainly used in the same way as the previously mentioned 2D games, we begin to see examples of weather being used as a part of gameplay such as in The Legend of Zelda: Ocarina of Time. These instances act more as superficial solutions to puzzles and are entirely scripted events that only have one designated effect on the game.

These two previous examples are more in line with a Static weather system. As game technology has advanced in the ongoing decades, we start to see the emergence of more video games using Dynamic weather systems. With the average graphical processing unit being able to run faster

and render graphics which were not possible in previous generations. Games such as Metal Gear Solid V: The Phantom Pain have been able to create weather systems that make unique gameplay scenarios possible. While playing in the game weather effects such as sandstorms can be generated, seemingly at random to the player. This can create seamless changes to gameplay such as reducing player and enemy visibility, organically causing the player to change the way they play and find a way to overcome new obstacles.

2.2 Key Technologies surrounding weather effects.

When it comes to the creation of weather effects in games the technology that the developers within the industry use fall under the umbrella term *Visual Effects.* These effects are used in many aspects of video game development, ranging from small-scale events such as sparks from a bullet ricochet all the way to large-scale events such as tornados. All of these rely on particle effects and are mainly differentiated by how developers adjust certain parameters and apply their own artistic expression to make each effect stand out on its own. Bernelin defines a particle effect as a physics simulation where we then put images on top of different points that a physically simulated to create the sense of giving a volumetric effect. (Kasurinen, Miller, Bernelind and Raitanen, 2019) This "sense" Bernelind refers to the fact that while most particle effects in 3D games appear 3D, the particles are rendered as 2D images which are then always orientated to be facing the game camera. Doing this helps to reduce the overall strain the particle effects have on the underlying hardware as the sheer quantity of particles would be too much for most conventional computers and consoles to render in real-time. Once the overall movement and behaviour of the particle are created, developers can then use 3D meshes and other elements to represent the visible element of the particle.

For weather effects such as droplets and snowflakes, this element can be a simple 2D image that will move and rotate around the scene based off the predetermined behaviour. Flipbooks are a form of animation that cycles through a given sprite sheet creating the effect of particles changing shape and evolving over time. Using the flipbook method is commonly used in the industry to help create effects that represent gases such as clouds and explosions which will naturally change volume and dissipate over time.

Other common tools that are frequently used in the modern games industry are Graphical Shaders. A shader is a program that runs within the graphics pipeline to determine how to render each pixel on screen. Original shaders were used to calculate how lighting would be rendered onto objects and how this would alter the base colour of game objects (Bailey and Cunningham, n.d.).

As they have advanced, shaders can now be used for a greater series of effects such as rendering animated materials onto objects and vertex displacement, which is the process of moving the points within a corresponding mesh. These newer uses for shaders are now widely adopted in the industry as they are frequently used to create believable water bodies of water.

As this project implements particle effects to display parts of weather like raindrops and snowflakes, using shaders alongside them will help create the impression of the particles interacting with the rest of the environment as the shaders will be built in a way that allows them

to have incremental changes in their appearance over time, as if the particles were accumulating over the whole surface.

2.3 Weather Interactions with a Game environment

The progression of weather effects in the last decade has come about through the advent of more particle effects which allow for more advanced behaviours with the environment. With these new effects, developers have been able to use them to help enrich their game mechanics.

When developing the VFX for Ghost of Tsushima, Vainio highlighted that the two main areas of improvement for the project's VFX compared to Sucker Punch Productions' previous game, Infamous Second Son, were improving the level of interactivity and building the particles at a larger scale (Vainio, 2021). To meet the game's main artistic goal of having everything on-screen move many particles (falling leaves, raindrops, and gusts of wind) were utilised alongside a global wind system that allows for all particles in the environment to behave in line with each other. The development team also decided to use this wind system directly into the gameplay and allowed players to manipulate these particles indirectly through the navigation system, as the wind system will change its direction towards the given objective and cause all particles in the scene to adjust accordingly.

2.4 GPU vs CPU rendering

When it comes to rendering graphics on a computer there are two types of hardware components on a traditional computer that can be used, the Central Processing Unit and the Graphics processing unit. Findings showed that the GTX280 GPU which they conducted their testing on, averaged data transfer times which were 2.5X than that of the Core i7-960 CPU (Lee et al., 2010). This most likely stems from the fact GPUs are composed of a larger number of low-power cores relative to that of the CPU. This means that it can do a larger sum of simple instructions which is what graphics-based rendering is largely comprised of. This highlights how choosing to render weather effects on the GPU is the optimal choice due to how much more efficiently it processes the relevant data compared to a CPU-based approach.

As games commonly utilise some form of visual particle effects to simulate types of weather such as rain, snow and even fog. As this would be a very significant part of the project, considering how either of the hardware components is able to render particle effects is an important factor when deciding what method of rendering for the system. Fortunately, some game engines such as Unity have tools that allow for particle rendering on both the CPU and GPU. These are the CPU-based Built-In Particle System and the GPU-based Visual Effect Graph (Technologies, 2019).

Feature	Built-in Particle System(BiP)	Visual Effect Graph(VFXG)
Render Pipeline compatibility	• Built-in Render Pipeline •Universal Render Pipeline •High Definition Render Pipeline	•Universal Render Pipeline •High Definition Render Pipeline
Feasible number of particles	Thousands	Millions
Particle system authoring	Simple modular authoring process that uses the Particle System component in the Inspector. Each module represents a predefined behavior for the particle.	Highly customisable authoring process that uses a graph view.
Physics	Particles can interact with Unity's underlying physics system.	Particles can interact with specific elements that you define in the Visual Effect Graph. For example, particles can interact with the depth buffer.
Script interaction	You can use C# scripts to fully customise the Particle System at runtime. You can read from and write to each particle in a system and respond to collision events. The Particle System component also provides playback control API. This means that you can use scripts to play and pause the effect and simulate the effect with custom step sizes.	You can expose graph properties and access them through C# scripts to customise instances of the effect. You can also use the Event Interface to send custom events with attached data that the graph can process. The Visual Effect component also provides playback control API.
Frame buffers	No	In the High-Definition Render Pipeline, provides access to the color and depth buffer. For example, you can sample the color buffer and use the result to set particle color, or you can use the depth buffer to simulate collisions.

Table 1: Comparisons between the available particle systems and their advantages/limitations

Both options for creating particle effects have definitive advantages. The BiP's ability to interact with the underlying physics system allows for a greater level of interactivity between the particles and other game objects within the environment. This kind of behavior would allow for the high-quality weather effects seen in Triple-A games such as Red Dead Redemption 2 (Rockstar Games, 2018), where particles such as snow respond to collisions with other game objects by changing their positions and allowing for more realistic behaviour.

Figure 2: Red Dead Redemption 2's weather simulations allow for real-time changes as a player navigates through the environment.

While the VFXG does allow for some physics interaction allow through the depth buffer, this method would not allow for the same level of interactions that would be possible with the BiP. However, the main advantage VFXG has is the "feasible" number of particles that it can render on screen. As it can render millions of particles compared to BiP's thousands of particles, this would make the GPU rendering the more suitable option for a weather system to create weather effects that can be convincing and be close to real-life weather effects.

3. Methodology

This section gives a description of the development process for the weather system and reasoning for the decisions made in this process.

None of the assets used in the implementation are my own work, all the assets are either provided by unity or were borrowed from online.

3.1 Development Approach

Upon deciding the project's aims and objectives, it was then time to decide which development methodology will improve the development of the weather system the best. Agile methodologies have the key benefit of adaptability. As the project covers a topic which I was largely unfamiliar with at the start, the benefits of using Agile could be applicable as deviations from the plan are considered normal and perceived to be beneficial for an overall greater product (Thummadi, Shiv and Lyytinen, 2011). The use of two weeks sprints to complete assignments is also seen as a proven method to help achieve the different objectives of a project by dedicating all that focus to one objective for that entire two-week period. However, these methodologies are often best utilised for team projects, as the practice of using work sprints helps to complete a predetermined list of deliverables assigned.

Waterfall methods allow for a sequential approach to software development. Following the waterfall approach allows the project to be organised in several ongoing phases. While the planned project timeline does allow for minor overlap between different tasks and stages of development, the overall timeline allows for ample time for tasks to be completed. Due to the external responsibilities outside the scope of this project. Using this methodology would be the ideal choice over the two discussed in this paper. This mainly stems that it can more realistically accommodate some of the time restrictions which come with the overall project as the sprint-focused approach of the agile methodology cannot be achieved due to the time required for each sprint.

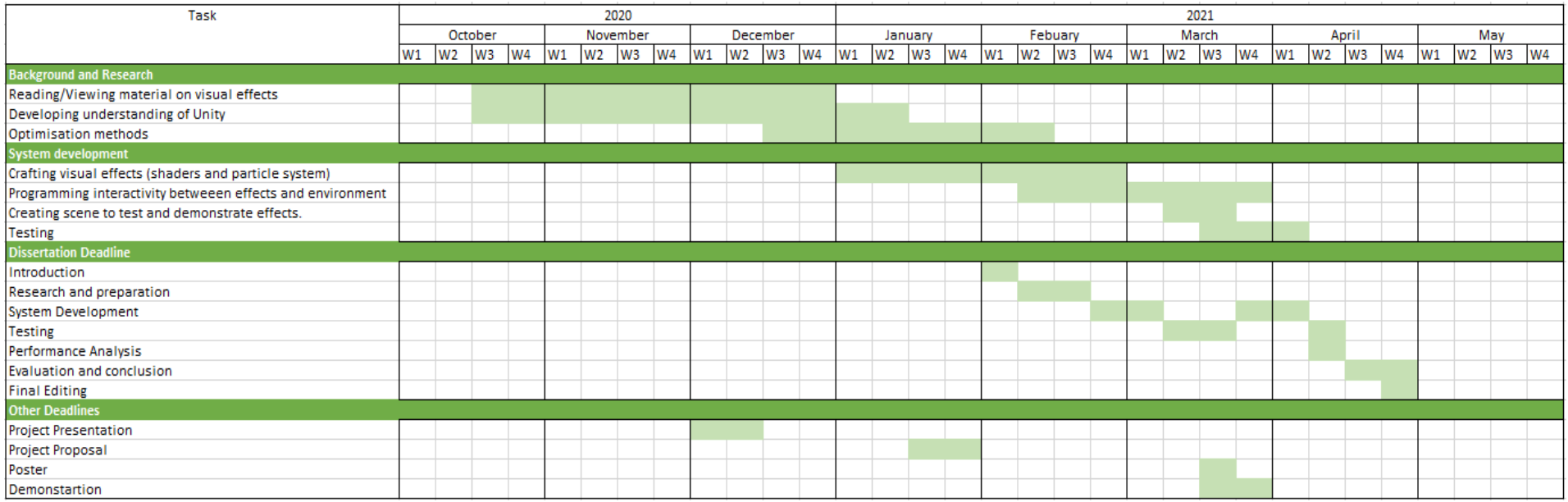

Figure 3: The Gantt chart which was used to map out each phase of this project over the seven month period using the Waterfall development approach.

3.2 Building Particle Effects

After considering the findings from the background on the benefits of using GPU rendering compared to CPU rendering, the decision was made to use GPU rendering via Unity's VFX Graph tool. This tool uses contextual nodes which can be connected to help streamline the process of building the particle effect compared to the BiP which does require scripting when constructing effects.

When constructing a new VFX graph there are 4 context blocks that are created by default which forms the basis of any effect, see Figure 4.

Spawn Context: This context determines the rate for how many particles the system will spawn at each call. This context works alongside the Initialize context as it holds the capacity for how many particles can be spawned at one time. If the maximum capacity is reached, then the spawn context will stop until particles that already exist in the scene are removed.

Initialize Particle Context: This context establishes the initial properties of each particle that is spawned each frame. Such properties may include, but are not limited to, the velocity, rotation or colour of each particle. To compare this with scripting within Unity, this block behaves in a similar manner to the Start function, as it is only called at the beginning of a particle's initialization.

Update Particle Context: This context determines incremental changes to a particle over its lifetime. It is called every frame over the particles lifetime and will continue to apply values to the specified properties. For example, if a designer wishes to have a particle experience drag and have its velocity lower over the particle's lifetime then this could be implemented using a Force block here in the Update Context. Later in this section, it will be shown how this context in combination with a Turbulence block was used in the snow effect to create sporadic movement under external forces.

Output Particle Quad Context: This context determines how the particle will be rendered at run time. While the Initialize and Update Contexts help determine particles' physical positions in a scene, this context determines the appearance of the particle in that position. This can be determined by multiple factors such as particles' size, transparency and is also where textures can be set to a particle.

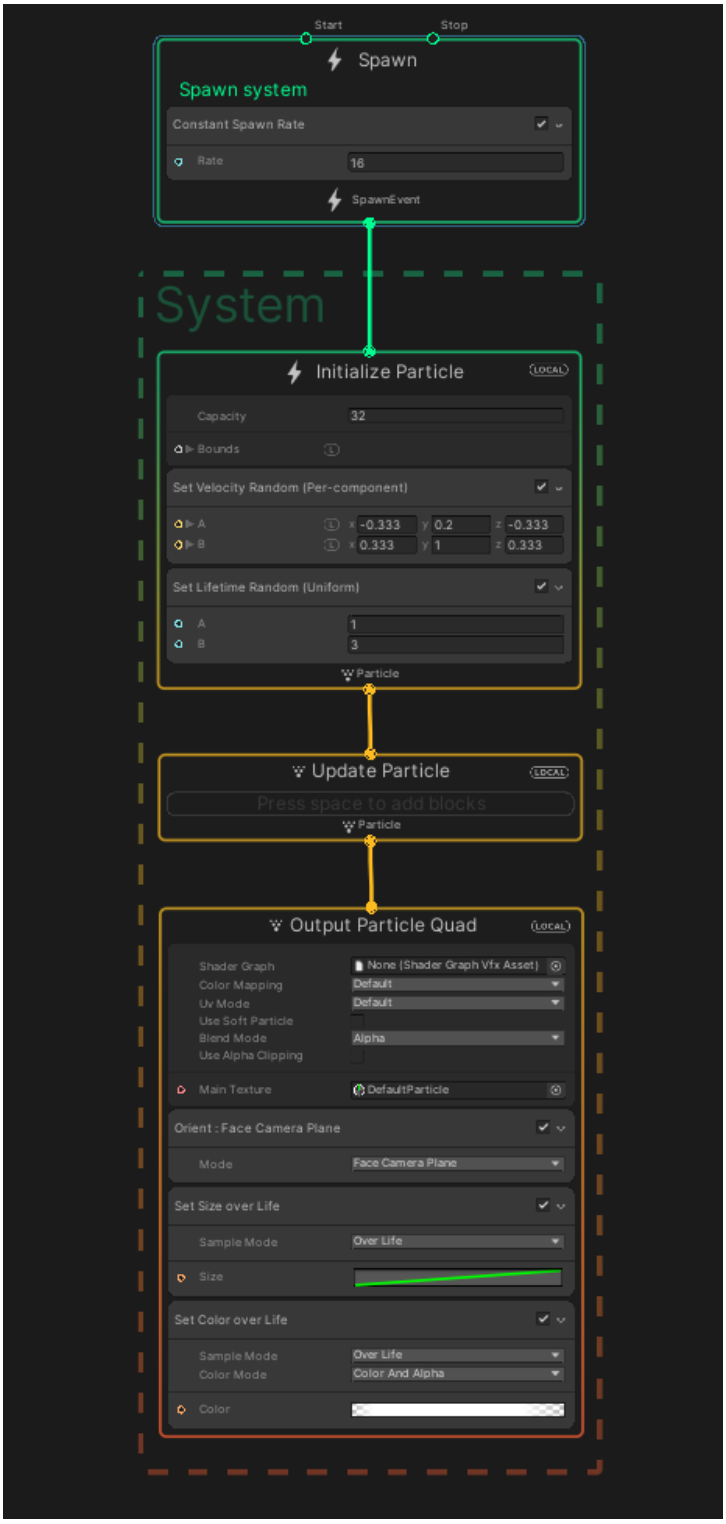

Figure 4:Default Contexts for the VFX graph

Every attribute block that is part of the VFX graph can be connected to an associating property of the same type. Every property associated with that specific VFX graph will then be stored in the systems Blackboard (Figure 5). This board allows the designer to change the respective values of these properties and for types such as floats can be set as ranges. By enabling the exposed boolean of these properties, they will be shown in the Unity Inspector, letting VFX artists adjust the look of their particles in the editor. The greater benefit of doing this is that this allows the properties to be manipulated through scripting allowing for dynamic behaviours such as by changing the total number of particles being spawned over time.

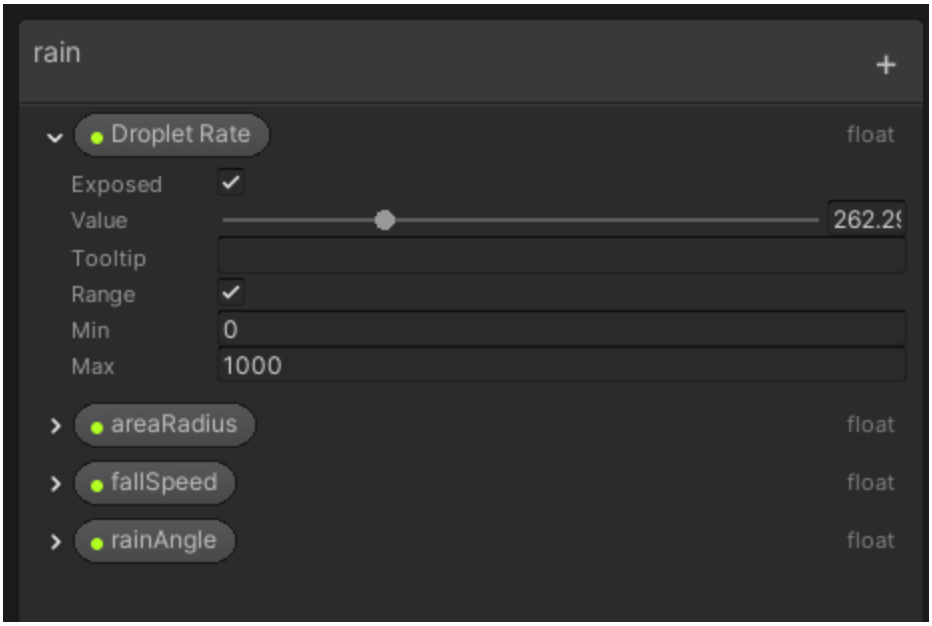

Figure 5: The VFX graph blackboard seen in the rain graph.

3.2.1 Creating the rain particles

The process of creating the rain droplet began by setting out the attributes for the particle's initialisation. The position block sets the spawn area for the particles in the shape of a cone, this allows the particles to spawn from a circular area that can spawn point (Figure 6). The particles' velocity is then set using exposed variables which help set the speed and falling direction that the particles are falling at.

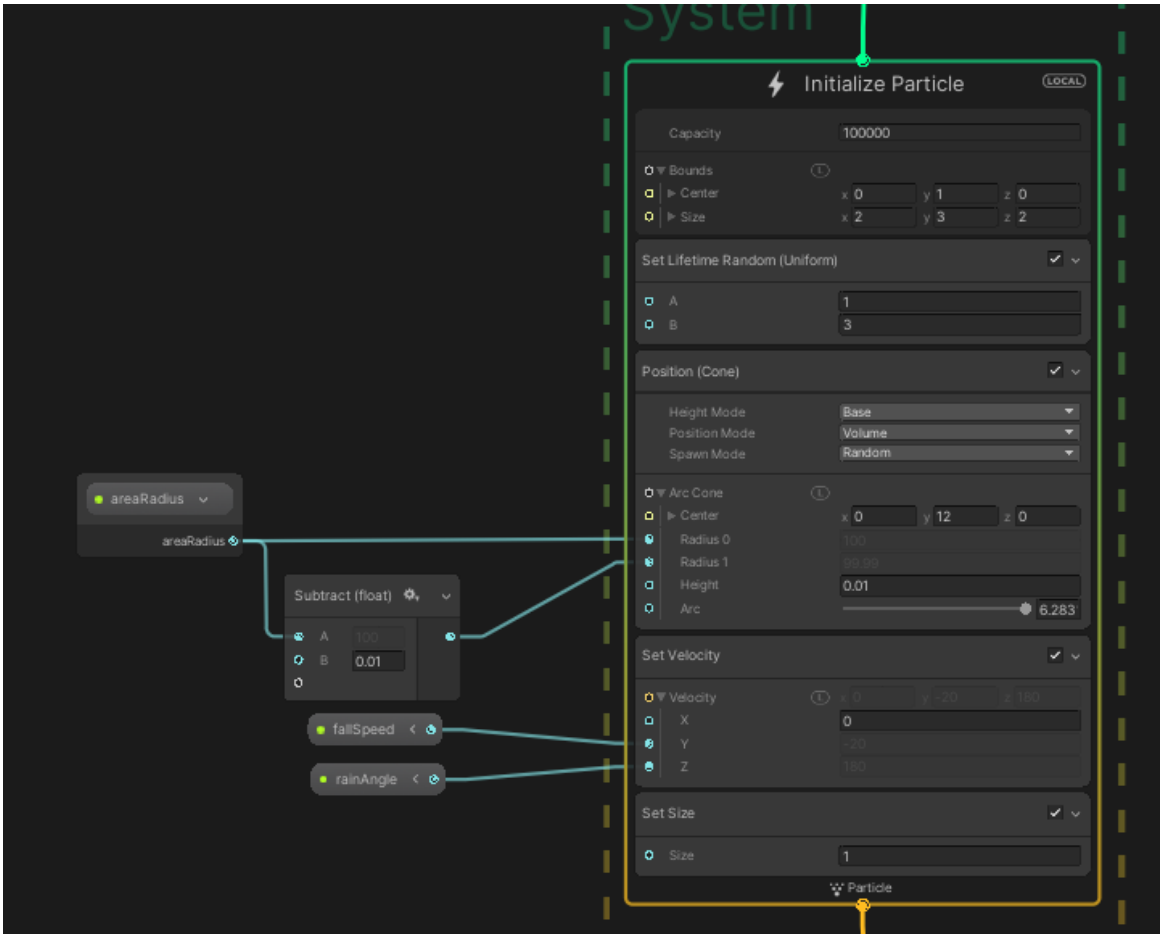

Figure 6: The Initialize context for the rain graph, this establishes the movement of the rain droplets from this particle effect.

With the movement of the particles created, the droplets needed to be given the appearance of the droplets being elongated by drag as they travel through the scene. This effect was achieved by creating a particle strip for each particle using a GPU event and was given a simple smoke texture that is then stretched as the parent particle travels through the scene (Figure 7). The GPU event allows for these new trails to be spawned based on the actions of the underlying droplet particle. For this specific system, the trigger event to spawn a singular trail for a droplet was set to the distance travelled each frame.

Figure 7: The GPU event which creates the falling trails for each individual rain particle.

3.2.2 Creating Fog Effect

To create a realistic fog effect, each particle spawned from this effect needed to have some animation applied to it to help give the effect the appearance of gas. This was achieved using texture sheet animation. The texture sheet in this case was a simple PNG file consisting of sixteen different gas images laid out in a 4x4 grid (Figure 8). By adjusting the Uv Mode to FlipbookBlend and setting the FlipBoook size to match the orientation of the grid, the VFX graph will then render each tile individually. Once a framerate has been set in a Flipbookplayer Block, the graph will then start cycling through each image of the texture sheet and creating the animated effect.

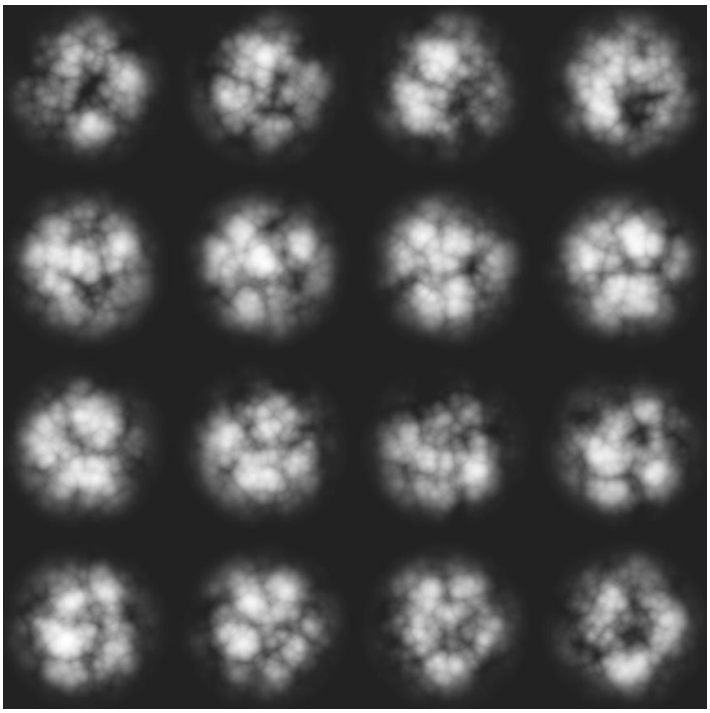

Figure 8: The PNG file for the fog effect which was used in the flipbook animation. (Flick, 2020)

As highlighted in the background section, there have been cases of games using their weather effects to influence the player's playstyle by obscuring their in-game view. In order to recreate this effect using the VFX graph the fog effect needed to interact with the lighting within a scene. To make this possible the default Output Particle Quad was replaced with an Output Particle Lit Quad and now allows for the effect with the same parameters to appear different depending on the kind of lighting a level designer chooses, creating an extra layer of customisation.

3.2.3. Creating Snow Particles

Unlike rain droplets, the snow particles in this project did not need to demonstrate any sort of deformation while moving through the scene as seen with the trails of the rain droplets. As snowflakes behave more as rigid bodies in the real world then it is acceptable to have each particle be rendered as a basic 2D sprite.

Where these particles are different from the rain droplets is that they needed to demonstrate more erratic movements as if they were interacting with wind and other forces while they descend onto the scene.

This movement was created using a turbulence block in the Update context. This block creates a noise field which applies random force to each particle periodically. The intensity property determines how strong this noise field is and thus also determines how much the noise block will influence the velocity of the particle. To have free control of the intensity in the editor, the Turbulent Strength property will directly link into the intensity channel.

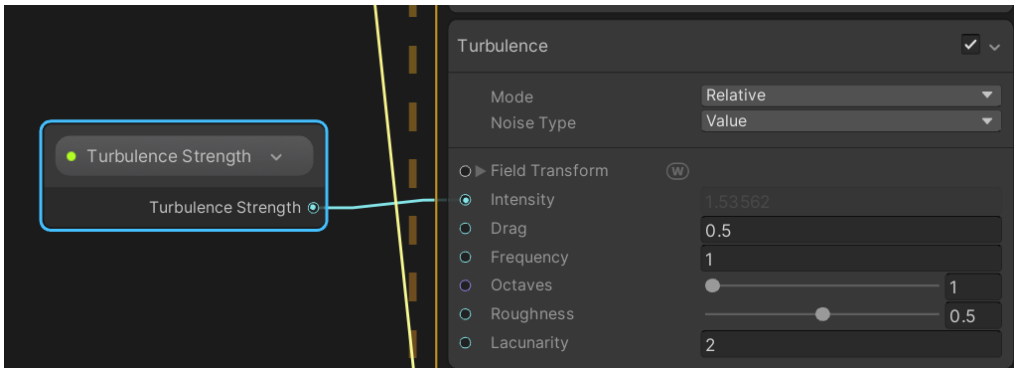

Figure 9: The turbulence field for the snow particles

As shown in figure 10, when the intensity of the turbulence is set at 0, the particles will in the direction of gravity set in the system. Despite the number of particles on screen, there is still some form of uniform spacing between each particle as they all obey a straight path down the y-axis. When the strength of the turbulence is increased, there is a clear change in the behaviour of the particles even though these still images. Any form of uniform spacing between the particles that was seen previously is now gone, as all the particles now follow their own unique paths within the system. It is this seemingly random movement that is being observed that was required to make the snow particles appear to behave more naturally.

Figure 10: The snow particles with the turbulence turned off (left) and then enabled (right)

3.3 Creating Shaders

In order to give the impression that the weather effects were creating changes within the scene that they were being spawned in, shader materials were used to help make changes to the overall appearance of objects within the environment.

These shaders were created using Unity's Shader Graph tool. This node-based system works in a very similar manner to the VFX graph, it even uses the same Blackboard interface to hold all the relevant variables that can be exposed and adjusted in the Inspector. The main difference is that the final shader is rendered through the PBR master node which processes all the different settings from the various connecting nodes (Figure 11).

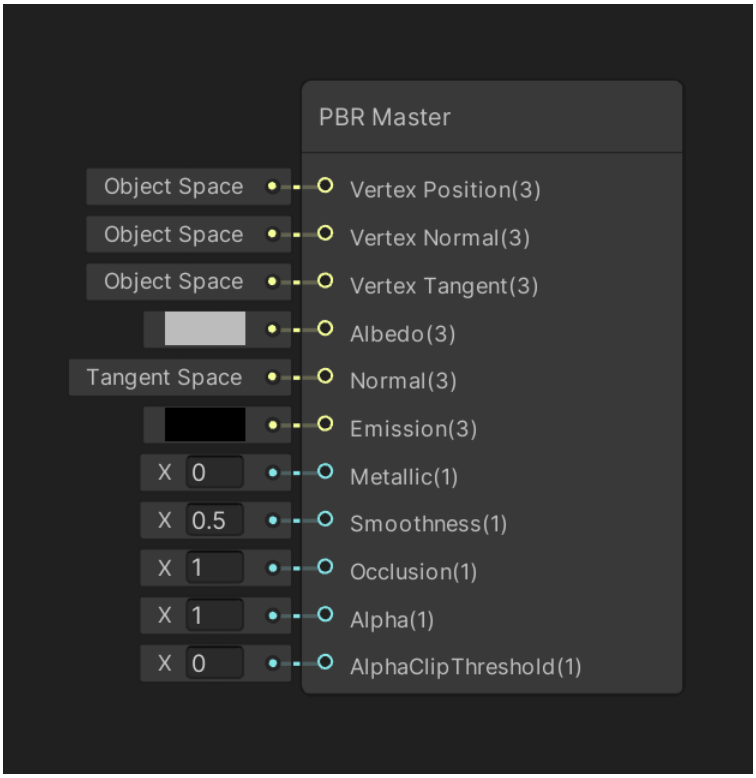

Figure 11:The PBR Master Node for Unity's Shader Graph tool

3.3.1 Creating Rainfall Shader

To create this effect of having the rain droplets impacting on a surface, texture sheet animation was utilised once again. As the ripple effect needed to blend correctly with any texture used for the materials albedo, the texture sheet was converted into a normal map. This ensures that the lighting being simulated accurately reflects off these ripples and renders the correct colours from the shader. If the texture sheets had not been converted, then the final ripples would not be as defined and not give enough visual appeal (Figure 12).

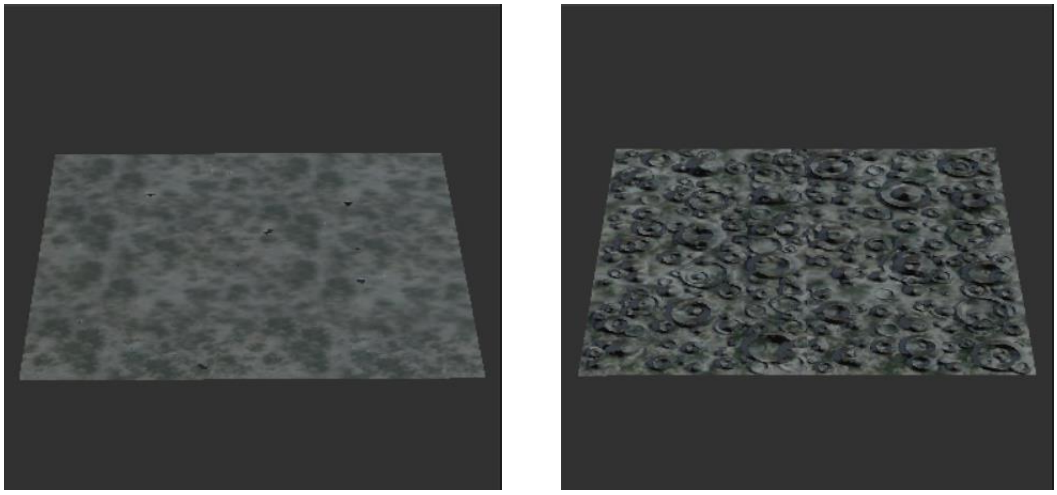

Figure 12: Differences between using the standard texture sheet on the shader (left) and the same sheet after converting it to a normal map (right)

Once the normal was applied, the flipbook animation could be created using the Flipbook node. Despite the visual differences between the interfaces, the process of creating the animation follows the same logic as the same kind of animation used in the VFX graph for the fog effect. The Flipbook node will read each image on the sheet and cycle through them to create the animated effect based on the Frame Per Second Property. To have this animation repeat over a given surface, a tiling and offset node will repeat this animation over a specified grid, completing the foundation for this shader.

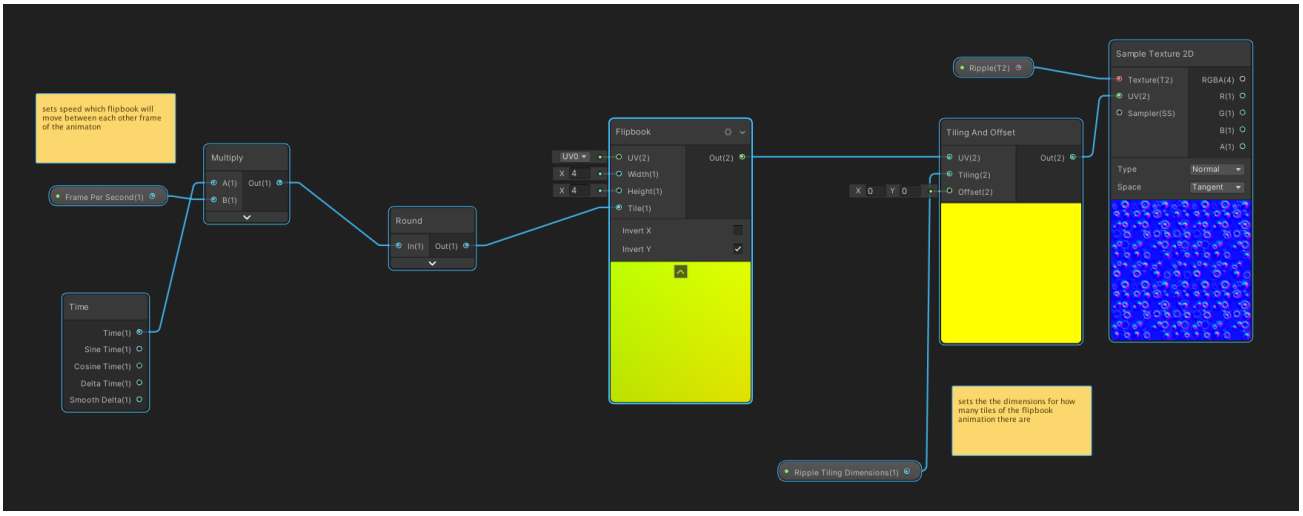

Figure 13: The section of the shader graph which creates a flipbook animation that can be tiles repeatedly over a given object.

However, while the base shader is running, the tiling becomes rather noticeable and no longer resembles the random falling pattern of rain in the real world. To hide this pattern, the nodes in figure 13 were duplicated except with a tiling grid of different size and were then distorted in order to shift the grid pattern. These two separate normals were then blended to create the final look of the shader (Figure 14).

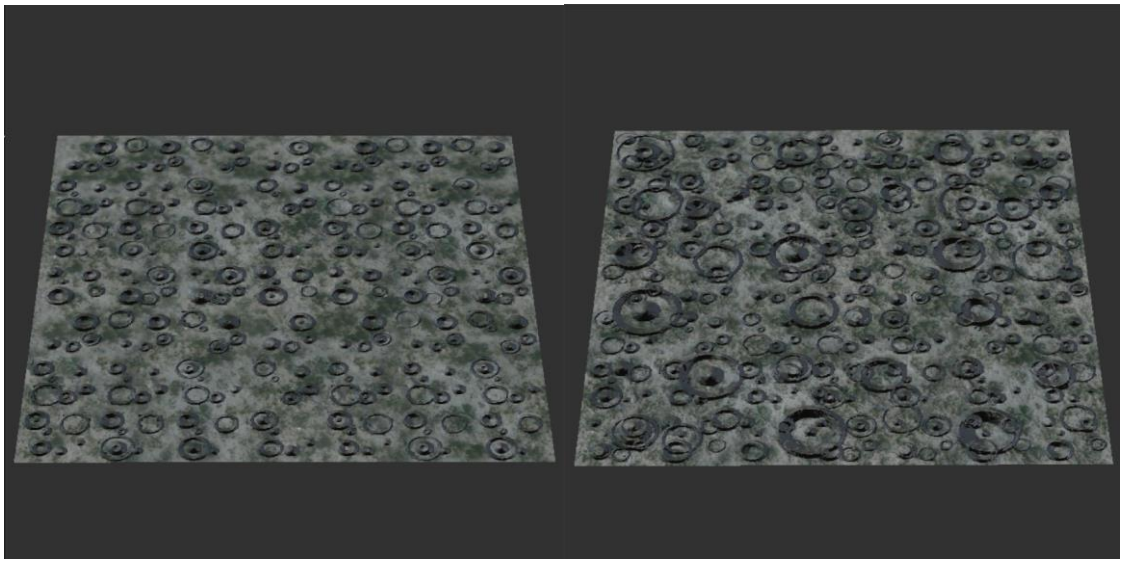

Figure 14: The rain shader with the initial flipbook tiling (left) in comparison to the shader once a second flipbook tile and distortion is added, obscuring any noticeable tiling in the material.

When rain is observed in the real world, there is often a natural escalation in the number of droplets falling at a given point. It is very rare to observe a heavy downpour appearing almost instantaneously and even in these cases we can observe a brief escalation. To make it possible for the shader to represent this escalation, the Ripple Density allows for changes in the visibility of the ripples on the shader. With this effect, lighter rainfall can be represented using a lower value for the density and through scripting, the strength of the rainfall can be programmed to scale with the number of particles being spawned in by the particle effect.

3.3.2 Creating Snowfall effect

To create a believable snow effect, there needed to be some way of having a shader that could increasingly cover the environment. By doing this, it would help simulate the effect of the snow particles landing on a surface and accumulating into separate snow piles. These piles would then need to be able to grow over time, eventually leading to them all merging into when pile.

The bulk of this shader's calculation is done by using several noise nodes. This helps determine the fall pattern of the snow, where the white sections of the pattern will be rendered as snow. The level of snow being rendered on the shader can be adjusted through the snow density property to determine how much of the pattern will be this white space while the Snow Blend Distance property will determine how much blending there will be between snow and the

underlying texture of the object the shader is on. These two values are then passed through as boundaries for a Smoothstep Node which uses Hermite interpolation in contrast to the Lerp Node which uses standard linear interpolation. This method of interpolation differs as it will speed up the rendering of the noise from the start and slow down as it approaches the end of the line it is interpolating (Ahlin, 1964). When applying this shader along with scripting this allow for smoother transitioning in the animation of the shader as the snow accumulates over time. Using this node will then help to determine what points of the noise pattern will be returned as a 0 or a 1, and which points will be blended between the two.

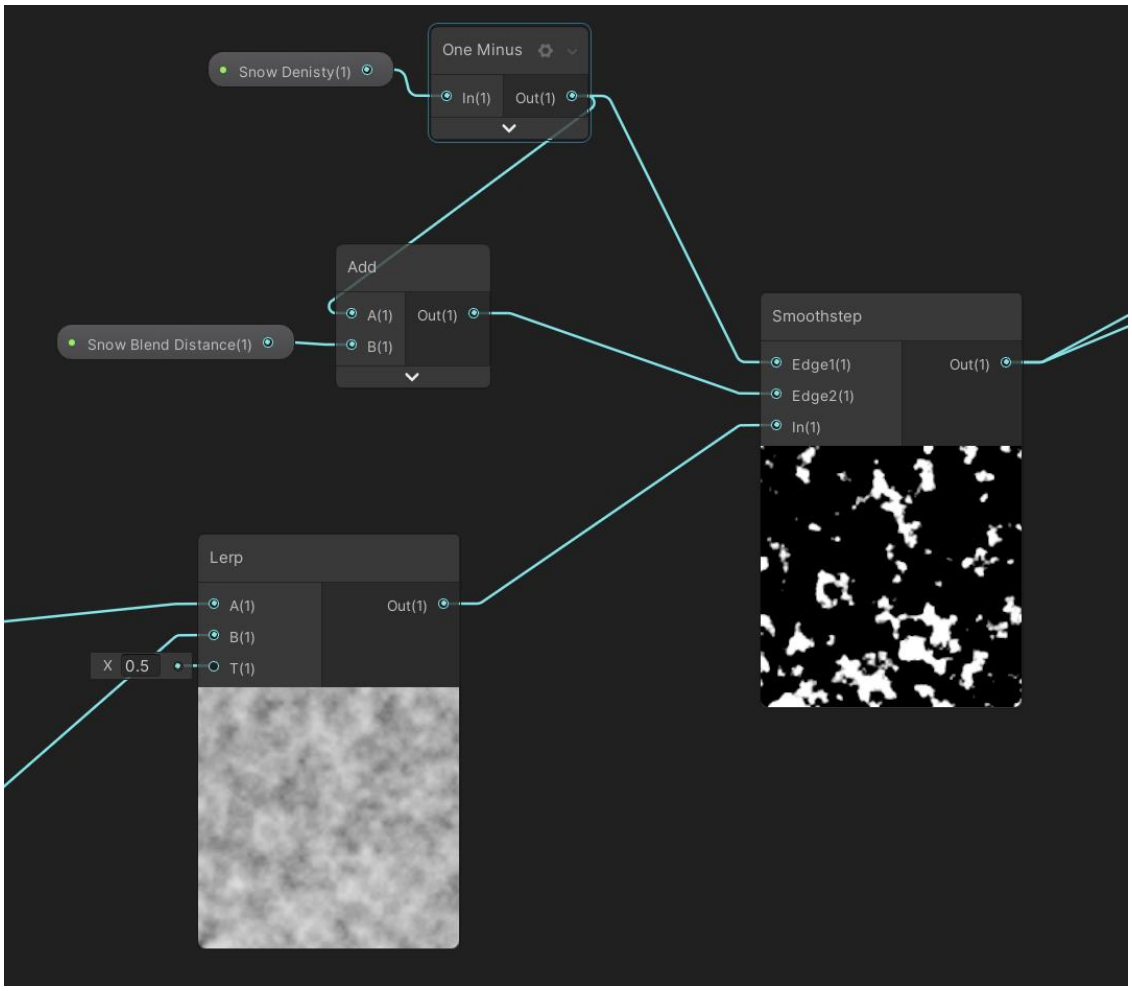

Figure 15: This series of nodes will calculate the noise field that will be applied over the texture at run time. Increasing the snow density variable will then increase the total amount of white area in the noise field.

In this example, the Snow Density and Blend Distance are set at 0.5 and 0.05 respectively. This means any value from the Lerp node that is less than o.5 will be returned as 0 and any over 0.55 is returned as 1 while all values in between are blended between 0 and 1 creating the final pattern seen in the smooth step.

By increasing the blend distance, a more natural blend between the snow and ground. However, having this value set too high creates a problem where the shader will consist almost entirely of

blending between the two textures and thus gives the shader the appearance of translucent snow which does not give off a realistic effect (Figure 16).

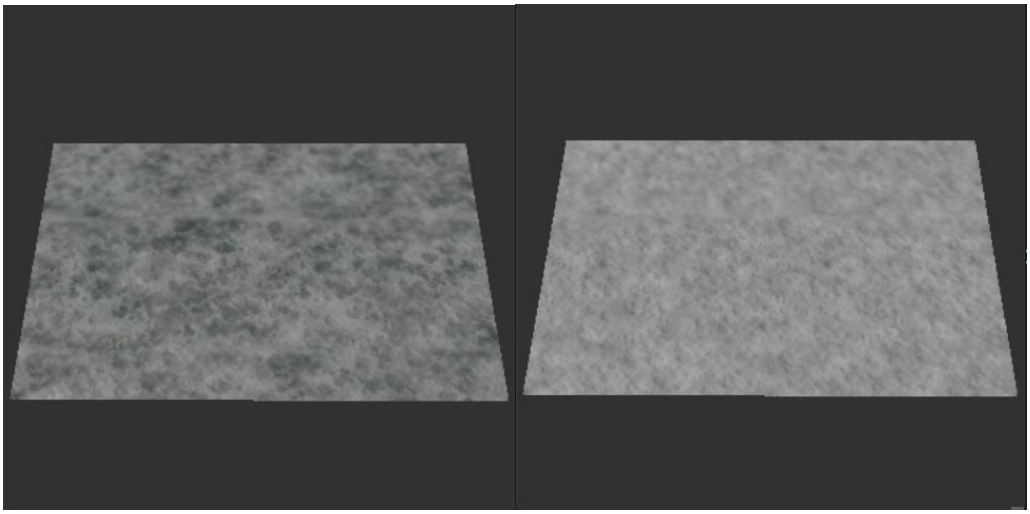

Figure 16:The snowfall shader with the snow density set at maximum in both. When the blend distance is set too high (left) the base texture can be seen through the noise, while when it is set in the ideal value (right) there is a distinct layer of snow that has accumulated on the ground texture.

3.4 Optimising the effects

With the main components of the weather system finished, the next part of the implementation focused on using various optimisation techniques with the main goals of:

- Improving the overall performance of the system while in play mode, the metric used to assess how well this has been achieved will be through monitoring the framerate while at run time.
- Reducing the cost of the system on the Graphical Processing Unit.
- Making changes to effects to help improve the efficiency of selecting the right parameters so the weather effects can match a game designer's desired look for a given effect.

3.4.1 Occlusion Culling

In an industry environment, games that feature a weather system such as the one created for this project would often have them covering a large section of a game's world. Given the sheer number of particles and effects that would need to be rendered at once for them to cover a whole section of a game world, this would become far too costly on the underlying hardware of any computer, let alone mainstream game consoles and PCs which are far more limited.

Unity already uses a solution to this problem known as Frustum Culling. This method reduces the total number of objects in a scene that the GPU must render by limiting it to the objects that fall within the boundaries of the game camera's field of view (Assarsson, Moller, 2000).

While frustum culling does help improve the performance of a gameplay scene, it has much room for improvement as it renders all objects that fall within the boundaries of the camera's FOV, even if they are placed behind other objects and not within the camera's clear line of sight. Fortunately, occlusion culling is a similar technique with the added advantage of being able to determine which geometry is hidden from the viewer by other geometry within the scene (El-Sana, Sokolovsky and Silva, 2001).

Unity does have its own version of occlusion culling that performs its runtime calculations on the CPU. This would be beneficial for this project as the effects used to create the system are being rendered through the GPU, which in theory means that using culling to reduce the number of objects for the game engine to render would create an overall benefit for the total performance of the GPU as it would require its own resources to carry out the culling.

While an attempt was made to implement occlusion culling into the final system, some unidentified issue with the Unity game engine has prevented the method from working as expected.

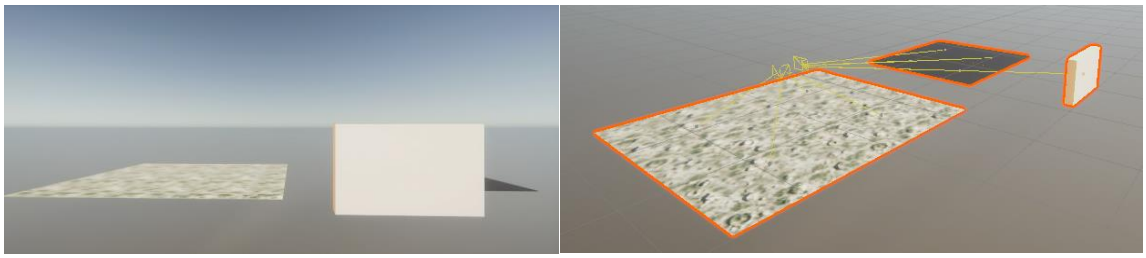

Figure 17: The setup of the scene to test the occlusion culling in the project. Each ground object is made of 4 joined planes which should each be culled individually when they leave the FOV of the camera or behind other objects.

The scene is set up with the block obscuring the four planes which have the snowfall shader attached to them (Figure 17). With occlusion culling, the two left most snowfall planes behind the block should not be rendered. However, the implementation shows that this did not occur when inspecting through the visualizer. The current view shows that game objects will not be rendered when they are out of the view of the camera as expected. This form of culling currently being presented falls more in line with the frustum culling discussed previously (Figure 18).

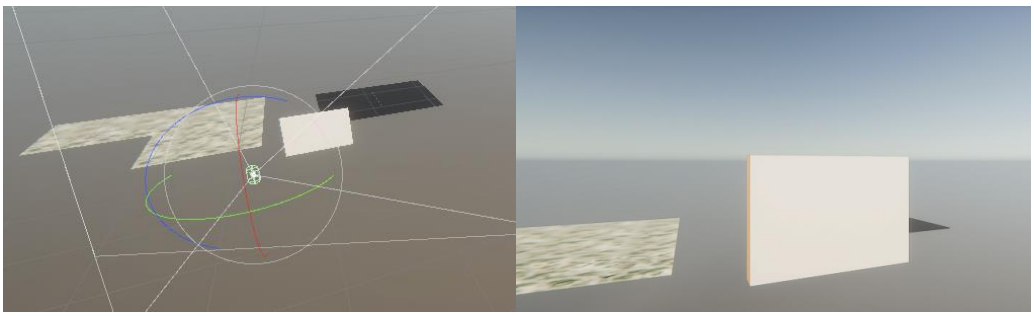

Figure 18: The test environment where the occlusion culling failed to work. As seen in the left frame, the culling does work properly when objects go out of the FOV but fail to ignore the black planes which are completely obscured by the wall.

Due to the time constraints of this project, this problem could not be looked at any further. While this came as a disappointment the underlying theory for occlusion culling makes a strong case for why it would be beneficial to include in any game that used the weather system due to the decreased cost on the GPU and by extension leading to an improvement in the overall frame rate of the project.

3.4.2 Orientating particles to the camera

The particle effects created for the weather system are the costliest component of the whole project. A significant factor to this is due to the number of particles that are present in the scene. While Unity's VFX Graph is capable of running millions of particles in a single instance, this will still become far more costly on the GPU's resources and so should be avoided where possible.

This problem arises when considering the large-scale game levels that would seek to use this sort of weather system. The first version of the particle effects, they were designed to be scaled by adjusting the overall spawn area for the particles and changing the spawn rate to achieve the desired particle density in the scene. While this approach would have worked for smaller levels with minimal draw distances, the system would become very easily overwhelmed once it was implemented in a large open world setting that would require the maximum number of particles that the VFX Graph could produce. The expected outcome of this would most likely be a greater strain on the GPU which would ultimately lead to lower frame rates with the potential for game crashes depending on the size of the scene.

To fix this issue, the visual effects needed to be altered. Rather than have each particle system spawn over any given area, the effects' positions were changed so they could be set to the main camera's position. This means that rather than having to set particle system to emit over a large area, a much smaller box can be created with the camera at the center and fewer particles will need to be emitted in the whole scene.

While this did help to reduce the total number of particles required to be emitted by each VFX graph, positioning them to emit around the camera required further changes to how the particles moved around the scene. The initial version of the particles had their speeds determined by a set velocity block. However, with the new setup the particles would always be falling in the same direction on the game screen no matter how the camera was rotated. To simplify, if the camera were looking to the horizon and showed the particles falling diagonally across the screen and was then positioned to look directly above its position, the particles would still move across the screen in the same direction rather than coming towards the camera which is what would be expected. This issue arose from the particles being set to move relative to the viewport of the camera. By removing the set velocity blocks and instead using a gravity block in the update context. This ensures that the particles fall relative to their position in the world and not relative to the camera's direction as it had done before (Figure 19).

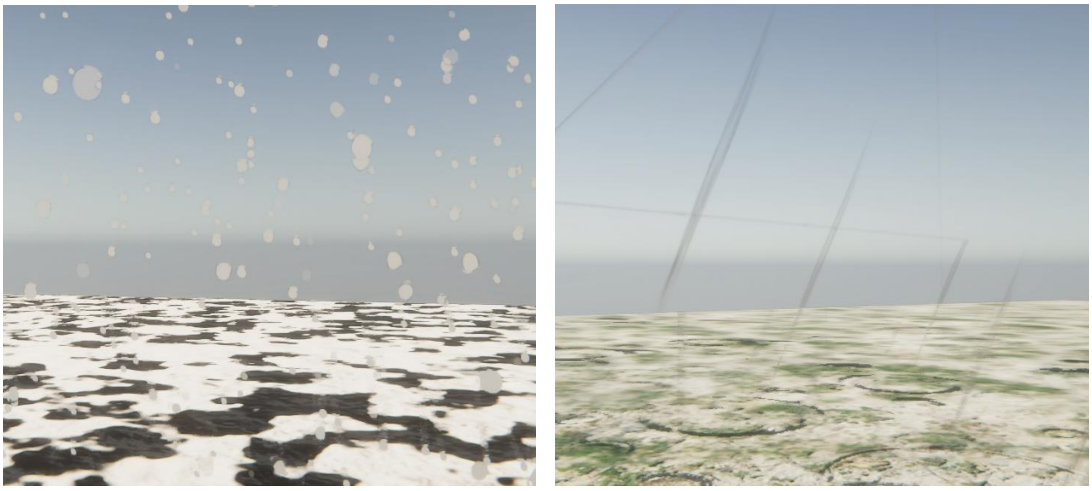

Figure 19: The two weather effects with the spawn position being set to the position of the camera.

4. Results

This section lays out the process used to collect the results of the weather system. It also provides the results and an analysis from these findings.

4.1 Outline of data gathering process

When addressing the outcome of the project, two areas must be considered. Firstly, a judgement of the behaviour of each weather effect needs to be made. The way each effect acts is a subjective matter, as every individual could judge the effects for not demonstrating certain characteristics that could be seen in other weather simulations or in real world examples. Because of this subjective nature, the behaviour will be assessed more on how closely the outcome is to the intended design and If they demonstrate any other characteristics that were not intended.

The second area that will be assessed is the actual performance of the system while being run and its cost on GPU. As a weather system is typically an additive feature, it is important that it is not too costly on the GPU as more important features such as rendering the geometry of the game world would need to take priority. This area of results also makes it possible to obtain more quantifiable.

When looking at the performance of the weather system, it is important to keep in mind that the specification of the components will be an important factor in how well the overall project will run. The computer that has been used for testing the performance and gathering results may be at a higher specification than the average computer on the market that may be used to run a game with this system included. Due to this factor, the results which are obtained from this project will not be a perfect representation of how this system will perform of other machines. However, because of the limitations of this project, it was not possible to test the performance of the weather system against an array of machines of varying levels of specification. This would have been ideal as it would have provided a greater range of data to use to quantify the success of the project.

Component	Specification
Processor	Intel® Core™ I7- 10750H CPU @ 2.60GHz 2.59GHz
Graphics	NVIDIA® GeForce RTX™ 2060 (6GB GDDR6 VRAM)
Memory	16GB Dual-Channel (8GB x 2) DDR4-2933MHz (15.9 GB usable)
Storage	M.2 NVMe Solid State Drive
Operating System	Windows 10 Home

Table 2: Specification for the Razer Blade 15 Laptop which was used to run and test the performance of the system

The relevant data on the performance of the weather system was obtained through the Unity Profiler (Figure 20). While the system is being run, the profiler will provide frame-by-frame feedback on the performance of each component of the underlying hardware, including for the purposes of this results analysis, the graphical processing unit.

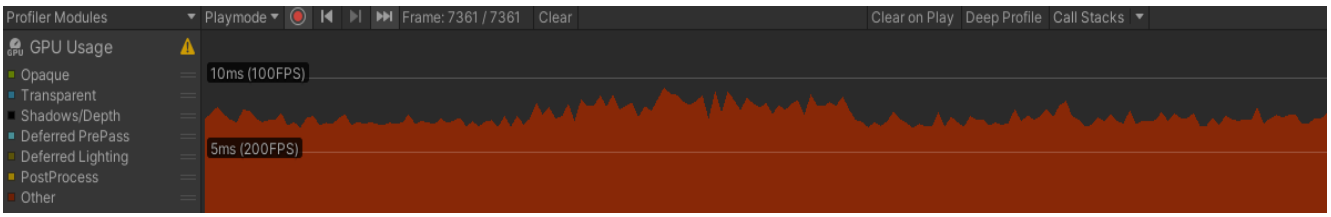

Figure 20: An example of the Unity Profiler showing the performance of the GPU at run time.

The profiler displays the performance of the selected component by outputting the total amount of time the component spent on each task for that frame. This time directly corresponds to the overall framerate that the system is performing at, with a smaller interval of time leading to a greater number of frames being displayed each second to the monitor.

While the profiler does provide a live feed of the performance of the underlying hardware, there is no official method to gain averages for these measurements provided by Unity. There is a Profiler Analyzer but that currently only provides data based on the CPU. Fortunately, this data can be accessed through scripting and be used to display the statistics directly into the game view.

Figure 21:UI display to show the statistics of the performance of the weather system.

4.2 The Rain Effect

As the rain effect is comprised of two main components, the particle effect and the animated shader, it is important to take several sets of readings on the performance of the system. Each set will be taken with different conditions in the game environment, which will be when;

- The shader and the particle system are both disabled.
- The particle system is enabled and emitting at a spawn rate of 1000 particles.
- The shader enabled, with the ripple effect at maximum density.
- The shader and particle system both enabled at maximised settings.

Metric	Reading
Default Scene	
Frame rate (Frames per Second)	116.5221
GPU Capacity (Mega Bytes)	5980
GPU usage (Percentage of capacity)	2.63
Particles Enabled	
Average Frame rate (Frames per Second)	104.9013
GPU Capacity (Mega Bytes)	5980
GPU usage (Percentage of capacity)	2.63
Shader Enabled	
Average Frame rate (Frames per Second)	82.7091
GPU Capacity (Mega Bytes)	5980
GPU usage (Percentage of capacity)	1.98
Both Enabled	
Average Frame rate (Frames per Second)	105.9011
GPU Capacity (Mega Bytes)	5980
GPU usage (Percentage of capacity)	2.63

Table 3: The benchmarking results for the rain system.

4.2.1 Defaults Rain Scene

Having readings with the rain effect completely disabled will provide a useful control for how the tested hardware handles running the basic Unity environment and provides the opportunity to show how much of an impact on performance each effect will have. The readings show that the computer used for results has very little difficulty rendering the default unity environment with the unanimated ground object as shown by the remarkably high average frame rate.

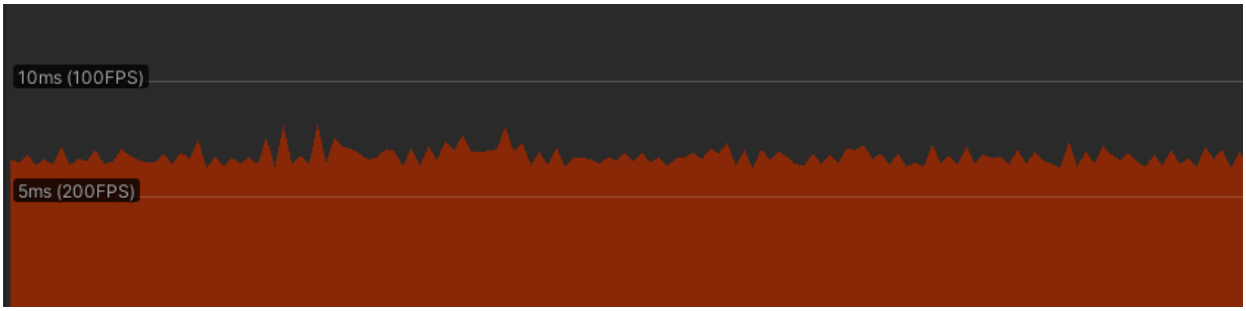

Figure 22:The profiler for the rain system's default scene.

4.2.2 Rain Particles Enabled

From the table, there is a noticeable drop in the average frame within the scene once the particle effect is fully enabled. This was not surprising, as the GPU now needed to create thousands of particles each second, so it is understandable to see this drop in performance.

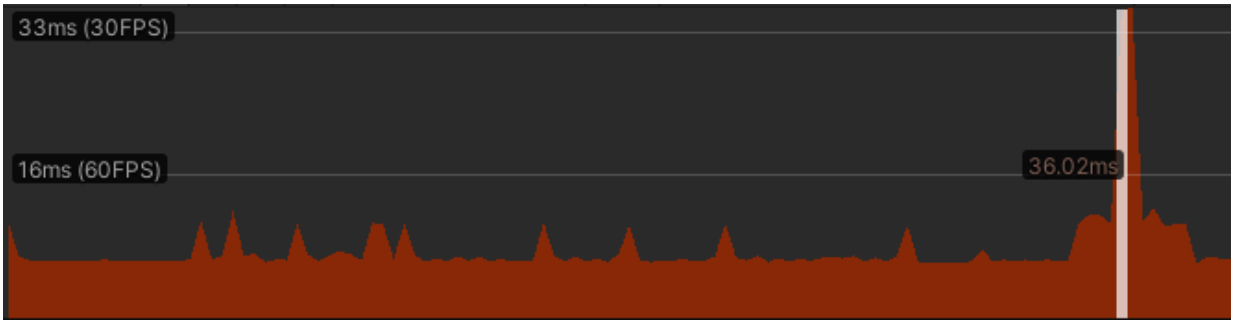

Figure 23: The profiler for the rain system's particles.

Looking at the profiler at the point the particle effect was enabled in the scene, it is quite clear that there was a significant drop in the performance of the project as the GPU began to calculate the behaviour for thousands of particles in an instant, more than doubling the amount of time taken on calculations. However, after this brief instance the profiler shows the GPU running at a level similar to those seen prior to enabling to particle effect. While this spike may seem like a point of concern at first, seeing as it is only present for a few milliseconds this momentary drop in performance is likely to go unnoticed to any human user and would not impede any sort of gameplay. This is further enforced by the fact that while the average frame rate did fall in this case, as it was still above the 100 FPS mark, which far exceeds the 60FPS used as a standard in the game's industry.

4.2.3 Rain Shader Enabled

With the shader enabled, there is a considerably larger drop in the performance of the GPU in comparison to enabling the particle effect, with respect to frame rate. When observing the profiler, the performance of the GPU is also shown to be far less stable, again compared to what was seen with the particle system. There is no clear reason as to why this is occurring except with the possibility that the scene consists of several plane objects which use the shader in order to demonstrate its uses over a large environment. As these are each separate objects running the flipbook animation independently, this could most likely explain why the performance of the GPU is fluctuating so much.

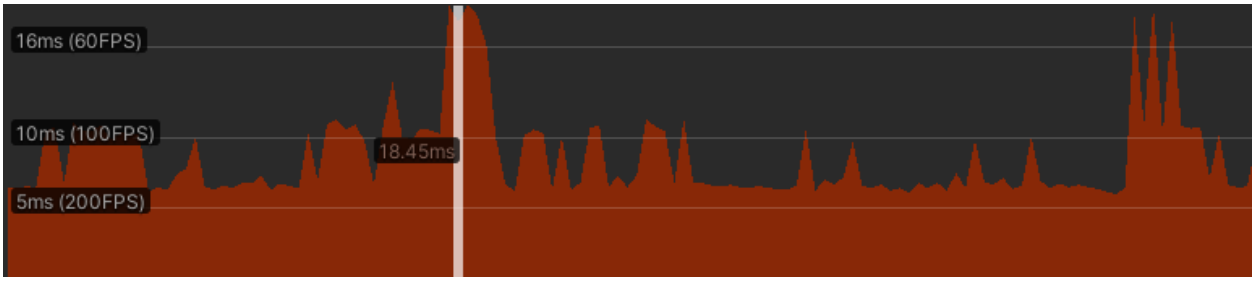

Figure 24:The profiler for the rain system's shader..

4.2.4 All Rain Components Enabled

With both effects for the rain system enabled, there is a noticeable drop in the frame rate of the scene when compared to the default scene and is performing at a similar level to when the scene just had the particles enabled. Interestingly, the frame rate at this point is still better when compared to when only the shader is enabled at full animation despite the GPU having an increased number of tasks that it must manage in the scene. When observing the profiler graph, the time the GPU is taking on each call falls in a much closer range (between 5 and 10 milliseconds). This shows that the previous result with just the shader could be an anomaly. Potential reasons for this could stem from the methods which were used to gather the results from the scene, such as the timestep used to calculate the average frame rate being too broad and creating a large margin of error from the correct frame rate. Another potential reason could be that the shader results were not taken in the same session, this may have created the inaccuracy within the results as Unity appears to require a different amount of the GPU's resources each time the system is run, even when no changes have been made to the scene. This difference in resources from the GPU may have potentially caused the unexpected pattern observed in these results.

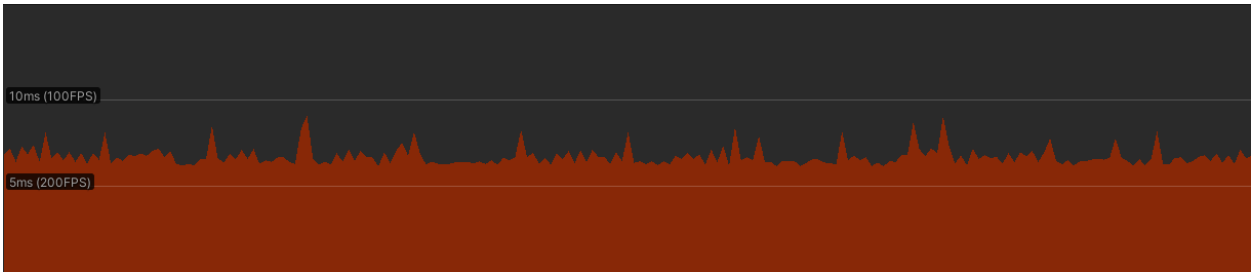

Figure 25:The profiler for all the rain system's components.

4.3 The Snow Effect

As the snow system has the same components as the rain system, being a shader and a particle system, the same conditions for gathering results have been used as the rain system. One unique aspect that has additionally been looked at is how the performance is affected when the density of the snow is increased over time.

Metric	Reading
Default Scene	
Average Frame rate (Frames per Second)	122.227
GPU Capacity (Mega Bytes)	5980
GPU usage (Percentage of capacity)	4.15
Particles Enabled	
Average Frame rate (Frames per Second)	113.7545
GPU Capacity (Mega Bytes)	5980
GPU usage (Percentage of capacity)	4.15
Shader Enabled	
Average Frame rate (Frames per Second)	122.7545
GPU Capacity (Mega Bytes)	5980
GPU usage (Percentage of capacity)	4.15
Both Enabled	
Average Frame rate (Frames per Second)	106.6926
GPU Capacity (Mega Bytes)	5980
GPU usage (Percentage of capacity)	4.15

Table 4: The benchmark results for the snow system.

4.3.1 Default Snow Scene

When looking at the results for the base scene for the snow system, the frame rate and status of the profiler are very similar and are performing at very close levels. A noticeable difference is how the snow system requires nearly double the number of resources from the GPU that rain system's scene. The greater requirement most likely stems from the fact that the plane representing the terrain uses a Lit Graph shader instead of a PBR graph. This type of shader allows for interaction with Unity's built-in lighting system which was needed to give the appearance of light reflecting off the snow. The added complexity of calculating these light interactions is therefore what has caused the extra cost on the GPU.

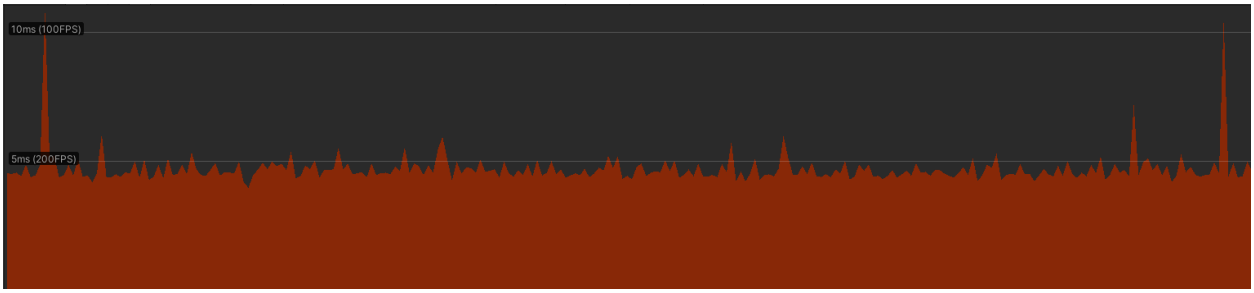

Figure 26:The profiler for the snow system's default scene.

4.3.2 Snow Particles Enabled

When the snow particles are enabled, there is an expected fall in the framerate. Looking at the results of the rain and snow particles, each lead to a similar drop in the framerate compared to the FPS of the default scene (a 9.97% and 6.93% fall respectively). With these findings, a fair assumption can be made on the impact of the particle effects on the performance of the scene, and that they will lead to an expected performance loss between six and 10 percent. However,

these findings can only be expected with the project running on a computer with specifications seen in table 4 or equivalent and running on a machine with a less powerful GPU would likely result in an even greater proportional fall in frame rate.

During run time, the profiler displayed a brief spike in time taken per calculation as seen in figure 27. The pattern shown was repeated by the profiler every few seconds, showing that it was most likely a persistent function that was being called. From the way the snow particles were built, the spike was mostly the VFX graph applying a new set of turbulence to each particle as it will persistently calculate and apply new particles every few frames.

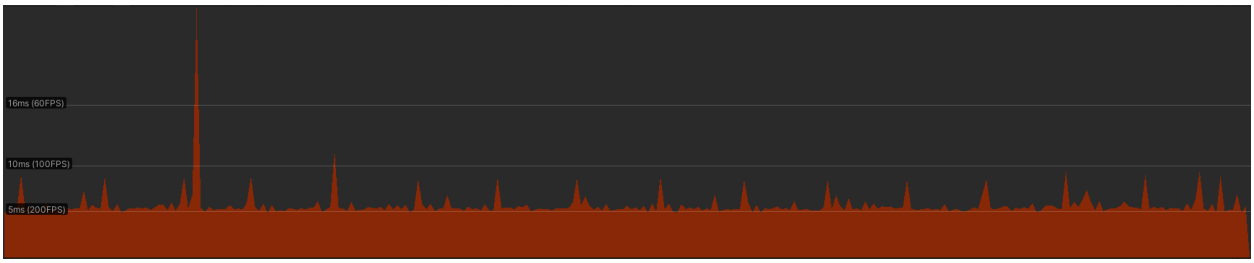

Figure 27:The profiler for the snow system's particles.

4.3.3 Snow Shader Enabled

Unlike with the rainfall shader, with the snowfall shader fully applied to the terrain the performance did not have any noticeable change and it in fact saw a marginal increase in the framerate. This outcome came about as the shader works by recolouring sections of a material white based on the noise pattern created by the shader. When the shader is set to its max density then the whole terrain will be rendered white with some detailing created using the normal map and so the amount of extra work that the GPU must do to render this is largely insignificant in terms of its impact on the performance.

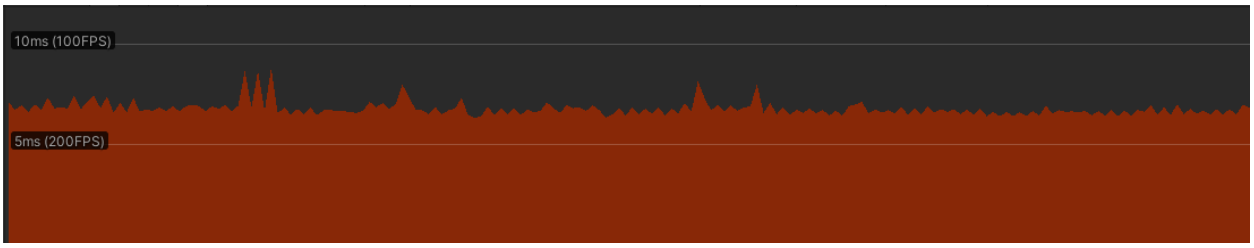

Figure 28:The profiler for the snow system's shader.

The key feature of this shader however is how can change the pattern of the snowfall over time. The program was run while gradually increasing the density of the snowfall at a constant rate. The results from the profiler show that this leads to the most varied and lowest frame rates seen in the whole project, even momentarily falling into the ten to twenty FPS range. This sort of frame rate would be considered below the accepted industry standard of 30 and 60 FPS however considering this example took place over a period of 20 seconds it is much faster than natural snow accumulation seen in nature. If a rate which was more closely matched to speeds found in real life were used, then this gradual change would likely not create such a dramatic decrease in the system performance.

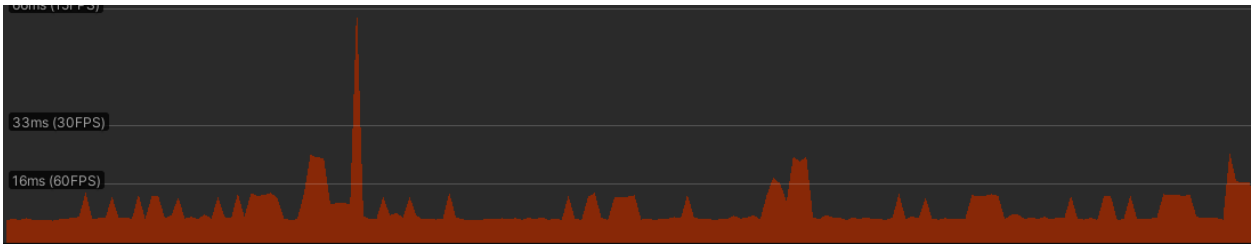

Figure 29:The profiler for the snow shader as the snow density is increased at a constant rate.

4.3.4 All Snow Components Enabled

The performance of the scene reaches its lowest average with both components of the snow system enabled at 106 frames per second. This sort of framerate is more what was expected when all components of a weather system are enabled as stated in the same test for the rain system. As the profiler for this test scenario present a similar pattern to when it was done for the rain system, this helps to support the assumption that the anomaly which was recorded was due to it being recorded in a different session of running the project within Unity.

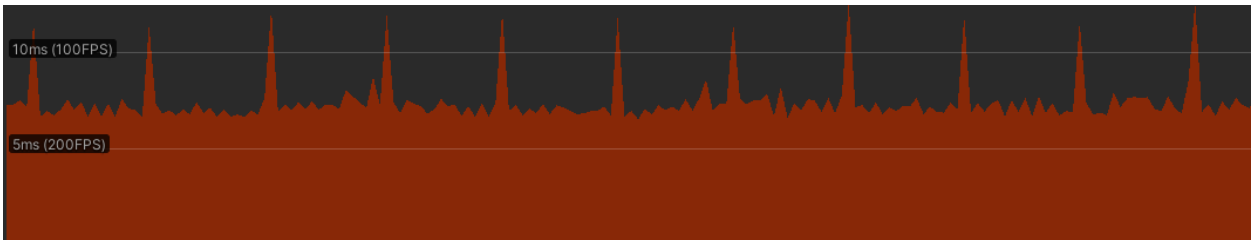

Figure 30:The profiler for all the snow system's components.

5. Evaluation

Judging the results gathered from the project, overall, they give strong evidence to show that the weather system that was created could have a potential implementation in an actual video game and not just within a basic scene with minimal environmental elements.

Apart from when the snowfall density was increased over a constant rate, every scenario which was tested provided an average frame rate above the 100 FPS mark. The industry standard for frame rates in games is 30 or 60 FPS depending on the genre. If the weather system in its present state were to be used in a game environment with all the essential assets such as non-player characters and level terrain, a framerate of 60 FPS would still likely be attainable. This theory is also enforced by the fact that all tested scenarios which had effects enabled, had them at maximum settings. Considering an in-game application for the weather system would see it used at various settings depending on the severity of weather designers want, then the system would likely not create as great a fall in performance as seen in the findings of this paper.

When looking at the most GPU intensive effects in the system (snow system) only required 4.15% of the GPU's total memory capacity. While at first glance this does not seem like a large amount of the GPU's resources being taken, the goal in a professional game is never to use 100% of the GPU's memory as this would have a negative impact on the system performance if it were to become too resource intensive. When running the system with a lower useable capacity in mind, the cost of the weather effect becomes vastly more significant and the values that were gathered may in fact show that this current version of the project may be too costly on the GPU once other game elements are added to the scene. The high cost of the system shows that the optimistaion techniques which were implemented did not have as significant an impact on lowering the requirements of the weather system as initially hoped. Further optimisation techniques, such as implementing proper occlusion culling, would most likely go very far in reducing the total cost on the GPU. The higher cost of the system on the GPU could also show that the application of the Unity profiler was not used as well as it potentially could have been to find what areas of the weather system were the most intensive on the GPU.

6. Conclusion

The final section provides final thoughts on the outcome of this project, what personal development was experienced throughout it and any work which can be done in any future versions of this project.

6.1 Satisfaction of Aims & Objectives

The aim of the project was to develop a weather system within the Unity Game's Engine that does require an extensive amount of the underlying hardware's processing power. With the final implementation, the success of achieving this aim can be assessed by comparing the implementation to the goals of this project.

To ensure that the main aim of this project was fulfilled, 5 key objectives were laid out.

1) *Study how industry developers implement weather systems and other particle effects into games available in the market.*

This objective was fulfilled in the background section by researching the history of weather systems in the video games industry and how the limitations of contemporary hardware influenced design decisions on what features could be implemented into those weather effects. It also provided knowledge on the tools and techniques modern-day developers use to create modern visual effects such as shaders, flipbook animations and Unity's VFX Graph.

2) *Develop an understanding of the Unity Game Engine.*

This objective was met through personal research and development which were conducted at the start of the project and through other game development projects which were done outside of this project and developed using Unity.

3) *Create a visual effects system that can simulate multiple types of weather effects.*

This objective was partially fulfilled as several visual effects such as particles and shaders were used to create a rain and snow system. To fully fulfil this objective, other types of effects could be created using the original two types as a base, e.g., building off from the rain system to create extreme forms of weather such as lightning storms and hurricanes.

4) *Design the effects to have interactive elements with other in-game elements.*

This objective was partially completed as the final implementation makes it possible to simulate a gradual increase in the impact of the weather effect o the environment as scripting can allow for the intensity of both the shader and the VFX graph to be increased over time. To fulfil this objective, features that were highlighted in the background section could have been implemented into the final project, such as interactions with the weather system and a player character that would cause damage to player health.

5) *Optimise the project so it does not become too costly on processing power.*

While some form of optimisation was added into the final implementation, some of the methods that were discussed did not manage to get implemented properly. As a result, the overall

performance of the project is not as optimised as it had the potential to be, this leaves the overall objective incomplete and is the area that would require the most attention on any future work.

Overall, the outcome of the project was partially successful in meeting the main aim of this project. The research phase of this project ensured a successful implementation of particle effects into the weather system, realistically depicting the same kind of behaviours that would be found in natural weather effects. While the individual effects simulate their own individual behaviour correctly, a greater amount of work could have been done to make the interactions between the effects and other game objects to create a greater level of depth as seen in modern games from the industry. Although there was an attempt to optimise the final system, more research and work could have been done to improve the performance and resource demands of the project.

6.2 Personal development

Throughout this project, I have developed my proficiency in several areas. With the research I conducted, I have greatly improved my knowledge in the field of visual effects and more specifically the area of weather effects. Firstly, the research has given me a history of how these effects were implemented into a final game. While in many cases these effects were used only to enhance the aesthetics of each game, several games in recent years have started to design their effects to have them interact with other systems within the game such as how extreme weather in Breath of the Wild can negatively affect player health. The research also gave me great insight into the technology developers use to create these effects. While I was aware of technologies such as particle systems and shaders, I had no prior knowledge on how to go about creating them and how to use them to create my desired effects.

At the start of this project, I had no prior experience in using a video game engine to develop a project for any purpose. By the end, I believe I have greatly improved my development skills. The project's focus on developing visual effects using shaders and particle systems has given me an in-depth knowledge of a technology that I believe can be used to help enhance any future projects that I may develop in Unity. Despite these two technologies being the primary focus of the project, the development process has helped me to gain a general understanding of how to approach game development projects and how to consider the areas such as optimization and testing from the very beginning as this will go a long way in improving the overall performance of my games.

6.3 Future Work

When looking at the outcome for the project there are several key areas that could be focused on to develop the project further:

- Developing shaders- The existing shaders in this project achieve an acceptable level of accuracy when representing their respective weather effects impact on the environment. However, creating a greater level of realism with these shaders could be possible with the utilisation of vertex displacement. This method would allow the shader to manipulate the 3D shape of the object carrying the shader. A potential improvement this would make on the current project is that rather than having the snowfall shader recolour portions of the mesh to give a 2D impression of snow accumulating on the ground, vertex displacement

would allow for a 3D impression of snow gathering into taller heaps over time.

- Using CPU-based particles- When looking at the creation of particle effects in this project, the VFX graph was chosen due to the greater number of particles that it can render in comparison to the built in particle system that Unity also provides. In the future, it may be worth looking at implanting CPU-based particles into the weather system. As these CPU particles can directly interact with the Unity physics system it would make it easier to implement more detailed interactions between particles and other objects within the environment. One potential feature, for example, would be having a player character slow down as they try to navigate through snow particles that have to accumulate on the ground.

  Due to the feasible number of particles the BiP can create is only in the thousands, this may not be a sufficient amount of particles depending on the scale of the game world. Another potential approach that could be taken is using a hybrid solution that uses the VFX graph to create the bulk of the particles in a scene to act as a visual element while the interactable particles from the BiP will only be spawned within a set radius from the camera's position.

- Implementing Wind- If future versions of this weather system implemented could involve some form of wind simulations, such as with Unity's Wind Zones. By building the particles with the consideration of having them be compatible with the wind zones would allow for a more consistent behaviour from all the differences that can be represented in the scene, as they will all have the same set of forces acting on them.

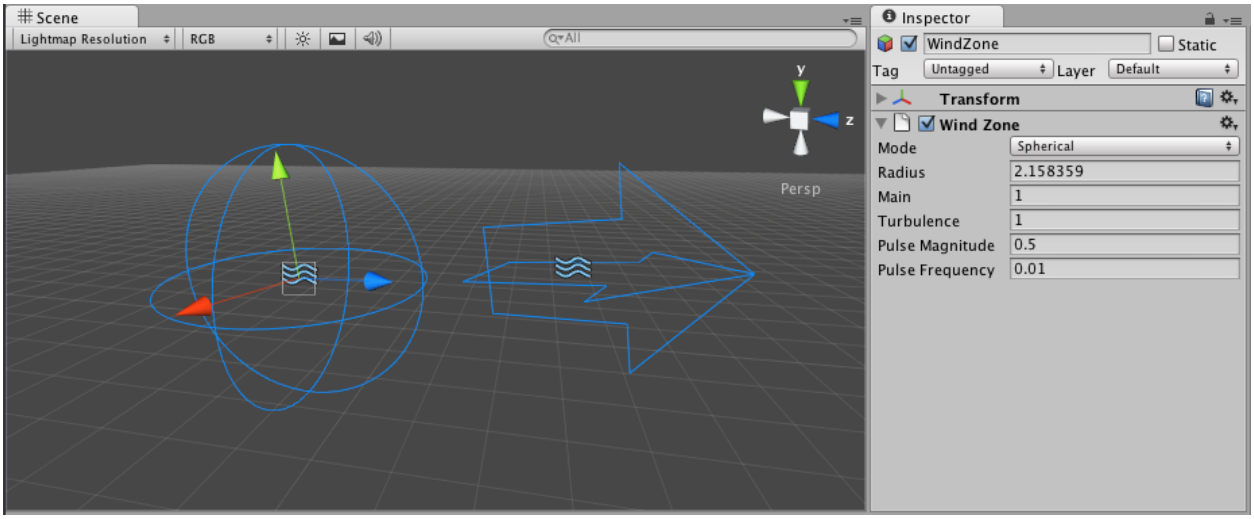

Figure 31:An example of Unity's Wind Zone system, which can be used to create a universal force within the scene to act on all the particles and objects within the system. (Technologies, 2017)

- Improving data collection of performance – When looking at the data that was gathered on the performance of the project, there were several points that could be improved on

that would not only make the results more reliable but also allow the data to highlight patterns in the weather system's performance. Firstly, it would have been better practice to ensure that all data gathering was completed in one session. This would have helped to reduce the risk of any anomalies from occurring, such as the one that appeared when gathering results for the rain shader, as it ensures that each set of data is collected with the same number of resources provided by the GPU.

Secondly, in any future work, there should be a focus to collect on the weather system using a wide range of machines with varying specifications. Repeating these benchmark tests on multiple machines will help recognise any patterns in how the performance will change based on how old or powerful the underlying hardware components are. While it would be a clear prediction to say that the performance will fall on older hardware, how much it is impacted by is also an important question to study. There could be some form of exponential trend in how much the performance of the weather system improves which each successive year's hardware. Discovering such a trend would help to evaluate the hardware considerations being made for developing such weather systems and when it is right to make alterations to certain resource-intensive features to accommodate lower end hardware and when some hardware should no longer be considered as it becomes too much of a detriment on the developers' vision for their system.

7. References

Ahlin, A., 1964. A Bivariate Generalization of Hermite's Interpolation Formula. *Mathematics of Computation*, [online] 18(86), p.264. Available at: <http://ams.org/journals/mcom/1964-18-086/S0025-5718-1964-0164428-6/S0025-5718-1964-0164428-6.pdf> [Accessed 8 April 2021].

Assarsson, U. and Moller, T., 2000. Optimized View Frustum Culling Algorithms for Bounding Boxes. *Journal of Graphics Tools*, [online] 5(1), pp.9-22. Available at: <http://www.cse.chalmers.se/~uffe/vfc_bbox.pdf> [Accessed 11 April 2021].

Bailey, M. and Cunningham, S., n.d. *Graphics Shaders*. 2nd ed. CRC Press.

Barton, M., 2021. *Game Studies - How's the Weather: Simulating Weather in Virtual Environments*. [online] Gamestudies.org. Available at: <http://gamestudies.org/0801/articles/barton> [Accessed 23 January 2021].

El-Sana, J., Sokolovsky, N. and Silva, C., 2001. Integrating occlusion culling with view-dependent rendering. In: *Proceedings Visualization, 2001. VIS '01.*. [online] San Diego, CA, USA: IEEE. Available at: <https://ieeexplore.ieee.org/abstract/document/964534> [Accessed 9 April 2021].

Flick, J., 2020. *Fog Texture Sheet*. [image] Available at: <Shadow.tech. 2020. The History of Gaming: The evolution of GPUs. [online] Available at: <https://shadow.tech/en-gb/blog/gaming/history-of-gaming-gpus> [Accessed 4 April 2021].> [Accessed 10 April 2021].

Gonzalez, J., 2021. *Maximizing Your Unity Game's Performance*. [online] CG Cookie. Available at: <https://cgcookie.com/ articles/ maximizing -your-unity-games- performance> [Accessed 24 January 2021].

Iche, T., 2020. VFX Graph - Building Visual Effects in the Spaceship Demo. [Blog] *Unity Unite*, Available at: <https://resources.unity.com/unitenow/onlinesessions/building-effects-with-vfx-graph-in-the-spaceship-demo> [Accessed 24 January 2021].

Jonasson, M. and Purho, P., 2012. *Juice it or lose it*.

Kasurinen, M., Miller, J., Bernelind, O. and Raitanen, E., 2019. *Video game explosions are actually a lie*.

Lee, V., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R. and Dubey, P., 2010. Debunking the 100X GPU vs. CPU myth. *ACM SIGARCH Computer Architecture News*, [online] 38(3), pp.451-460. Available at: <https://dl.acm.org/doi/pdf/10.1145/1815961.1816021> [Accessed 6 March 2021].

Nijman, J., 2013. *The art of screenshake*.

Nintendo (1991). *The Legend of Zelda: A Link to the Past* [Computer Game].

Nintendo (1994). *Super Metroid* [Computer Game].

Rehnberg, O., 2021. *Static vs dynamic weather systems in video games*. Undergraduate. Malmö University.

Rockstar Games (2018). *Red Dead Redemption 2* [Computer Game].

Shadow.tech. 2020. *The History of Gaming: The evolution of GPUs*. [online] Available at: <https://shadow.tech/en-gb/blog/gaming/history-of-gaming-gpus> [Accessed 4 April 2021].

Technologies, U., 2017. *Unity - Manual: Tree - Wind Zones*. [online] Docs.unity3d.com. Available at: <https://docs.unity3d.com/560/Documentation/Manual/class-WindZone.html> [Accessed 5 May 2021].

Technologies, U., 2019. *Unity - Manual: Choosing your particle system solution*. [online] Docs.unity3d.com. Available at: <https://docs.unity3d.com/Manual/ChoosingYourParticleSystem.html> [Accessed 12 February 2021].

Thummadi, B., Shiv, O. and Lyytinen, K., 2011. Enacted Routines in Agile and Waterfall Processes. In: *2011 Agile Conference*. [online] Salt Lake City, UT, USA: IEEE. Available at: <https://ieeexplore.ieee.org/abstract/document/6005487/authors#authors> [Accessed 9 April 2021].

Vainio, M., 2021. How stunning visual effects bring Ghost of Tsushima to life. [Blog] *Playstation.blog*, Available at: <https://blog.playstation.com/2021/01/12/how-stunning-visual-effects-bring-ghost-of-tsushima-to-life/> [Accessed 23 January 2021].