The Generation of Weather Systems in Video Games

Michael Rumley

150251891

MComp Computer Science

Supervisor: Dr Richard Davison

Abstract

The main objective of this project is to design and implement a way of rendering a weather system in a test scene.

This report details the design and implementation of this system, firstly by researching existing methods and then designing my own implementation. I detail the options available to me and justify the design decisions that were made.

The test scene I have delivered satisfies the projects objective, and suggestions are made for improvements to it. The final system has been fully tested and performance results are presented and evaluated.

Declaration

“I declare this dissertation represents my own work, unless otherwise stated”

Acknowledgements

I would like to thank Dr Richard Davison, Dr William Blewitt and the Newcastle games lab for all their support and assistance throughout this project.

CONTENTS

CONTENTS

Part 1: Introduction

1.1 Motivation and Rationale

In video games, the ultimate goal is to create a situation where a player is completely immersed in the world of the game. The inclusion of a convincing weather simulation is a major factor in contributing to this; the addition of weather makes a game more fundamentally 'real' by adding a degree of uncertainty into the game world. Whilst weather effects such as fog speed up render time by reducing the field of view, and snow can be modelled as a basic particle system with little augmentation needed, falling rain still lacks realism, probably since rain is a complex phenomenon with several audio and visual cues. This is despite being one of the most common weather conditions both in real life and in game scenes.
In developing a system to model rain, game developers have had to decide between speed and realism, with speed often being prioritised. For example, in early games able to show weather effects, the rain would be represented as an extra graphical layer over the regular level. Whilst the look and sound of the rain does help to build immersion, the effect is very basic, and has no effect on the characters, scenery or gameplay. This is because there would have to be extra animations made for each stage of a character's wetness, and the hardware limitations of early game systems would make this unattainable. However, as graphics hardware increases in performance, developers no longer have to consider this trade off so heavily, and games can now render complex weather systems in real time. Indeed, in modern gaming with AAA games running on powerful consoles and high- end gaming computers, the weather conditions not only look and act how a user would expect them to, but also have a tangible effect on the game world and the gameplay.

1.2 Aim and Objectives

The overall aim of this project is to investigate, implement and evaluate an approach to rendering rain in real time. Through researching existing systems, I shall compare different ways in which rain is rendered in games, before presenting an implementation of a suitable system for rendering rain and evaluating its performance and the conditions under which it performs best. This aim shall be accomplished through 4 main objectives.

1. Investigate existing approaches to rendering rain in video games
2. Select an appropriate graphical framework to render a rain test scene
3. Implement the scene with the framework
4. Evaluate the performance of the scene when parameters are changed and effects applied

The number of existing approaches I will investigate will vary based on the amount of time I have, researching will be an ongoing process that will run in parallel to developing my own approach, and so the number of approaches featured in the comparison will be dependent on this.

There are many open source libraries that can be used to render 3D graphics. The framework I choose will need to be extensive enough to render the rain test scene and include all the features and effects that need to be contained in it.

Testing of the scene shall compare my system to the approaches researched earlier, as well as the performance of the scene when aspects of it are changed and effects are applied to it.

Objective 1 will be discussed in further detail in section 2(Research), Objectives 2 and 3 will be covered in section 3(Design/Implementation) and Objective 4 will be featured in section 4(Results/Evaluation).

The timeline for this project is as follows:

27/10/17	Ethics Approval Form
1/11/17	Select graphical framework and familiarise myself with its usage (Objective 2)
3/11/17	Project Presentation
1/12/17	Project Proposal
13/4/18	Implement scene within framework (Objective 3)
20/4/18	Project Poster
1/5/18	Evaluate performance of scene (Objective 4)
4/5/18	Final Dissertation hand in
9/5/18	Poster/Demonstration Event

1.3 Paper Structure

Introduction

This section will introduce the project and explain the aims and objectives.

- Motivation and Rationale
- Aim and Objectives
- Paper Structure

Background Research

This section presents research into the project area

- Weather Effects
- Graphical Frameworks
- Modelling Techniques

Design/Implementation

This section describes the steps I took in designing and implementing the test scene

- Decision to use OpenGL
- NCLGL
- 3D Landscape
- Particle System
- Shaders
- Renderer
- User Input

Results/Evaluation

This section presents and analyses the results, and compares the system to the way it was intended to be designed

- Effect of Number of Particles on Frame Time
- Visual Inspection
- Evaluation of components

Conclusion

This section assesses whether the project has been successful.

- Fulfilment of Objectives
- Personal Reflection
- Further Work

PART 2: BACKGROUND RESEARCH

2.1 Weather Effects

One of the best ways to increase the level of realism found in a game world is by featuring convincing weather simulation. Weather systems not only make a game more graphically realistic, but more fundamentally real as they represent the unpredictable entering the structured and controllable game loop [1]. Implementing one of the most widely encountered weather conditions [2] into a 3D virtual environment would enhance the scene's realism and make it more believable to the player. The number of different techniques for rendering rain presented in background literature make it a strong, interesting candidate for further investigation.

2.3 Graphical Frameworks

There are many 3D graphics APIs that could be used to implement the rain test scene, such as X3DOM for internet applications [3], or Metal for iOS apps [4]. The main 2 APIs for rasterising graphics are DirectX [5] and OpenGL [6].

2.3.1 DirectX

DirectX is developed by Microsoft and is a collection of different APIs covering sound, input, networking and graphics. DirectX is used in the development of programs for Microsoft products- Windows computers and the Xbox family of systems. The latest version of DirectX is Version 12, released in 2015.

2.3.2 OpenGL

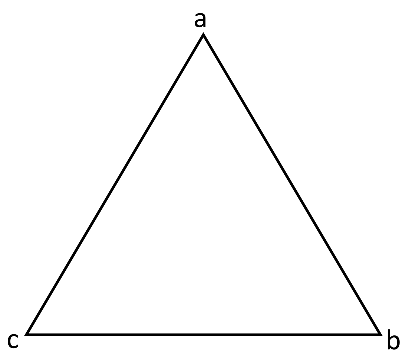
OpenGL was first developed in 1991 by Silicon Graphics inc, and since 2006 has been managed by the Kronos Group. Unlike DirectX, it is cross platform, and can be used with Linux, Android and PlayStation [7]. The latest version of OpenGL is Version 4.6, released in 2017.

2.4 Modelling Techniques

When deciding how to represent various real-world objects in a computer program, many different methods can be used. As computers become more powerful, more efficient ways of representing complex objects with many vertices have emerged.

2.4.1 Surface Based Modelling

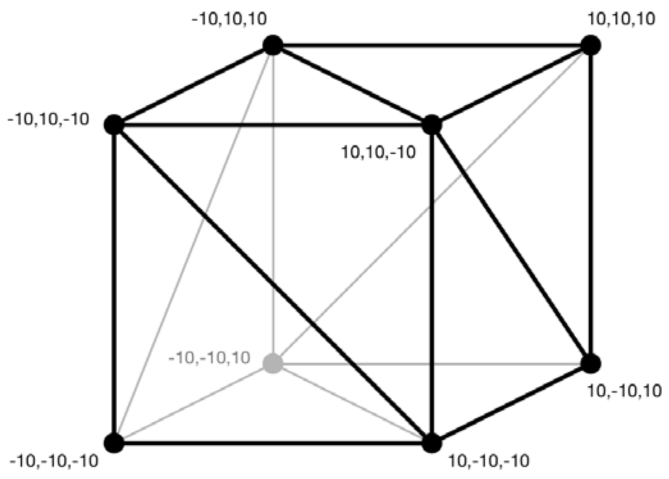
Surface based modelling is when an object is represented as a set of surfaces. Polygons can be built up using the basic geometric primitives provided by OpenGL (Points, Lines, Quads and Triangles) and then scaled appropriately to build complex objects. When drawing a triangle or a quadrilateral, a list of vertices is read and a shape is generated.



a (0,1)

b (1, -1)

c (-1, -1)



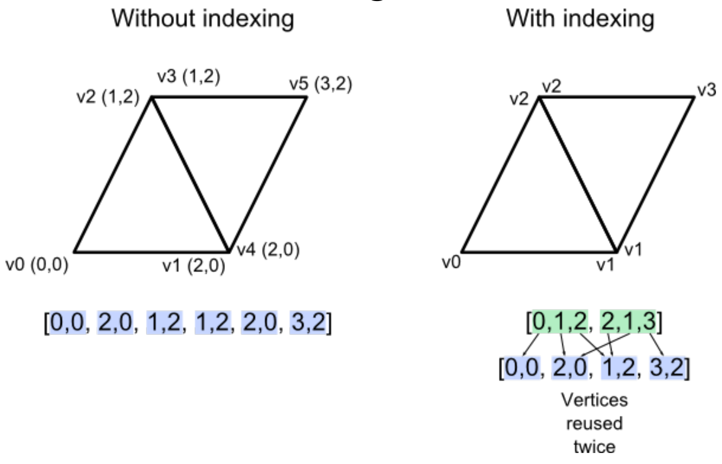
A cube made of triangles {1}

This technique is very powerful, as by increasing the number of triangles (or quadrilaterals) rasterised and changing their size accordingly, wireframes can be built up and complex shapes can be represented.

A witch character made up of hundreds of triangles {2}

However, it can be computationally very expensive to render a scene containing this many triangles, and as such the terrain for the test scene shall build upon this method using Index Buffering.

2.4.1.1 Index Buffering



An example of how index buffering reduces the number of vertices needed {3}

An index buffer acts as an array of pointers inside the buffer of vertices. This allows for the list of vertices to be reordered, and crucially, the same vertex to be reused. Previously, the number of vertices was equal to the number of indices, whereas using an index buffer has reduced this by 50%. As more triangles are required, this benefit increases.

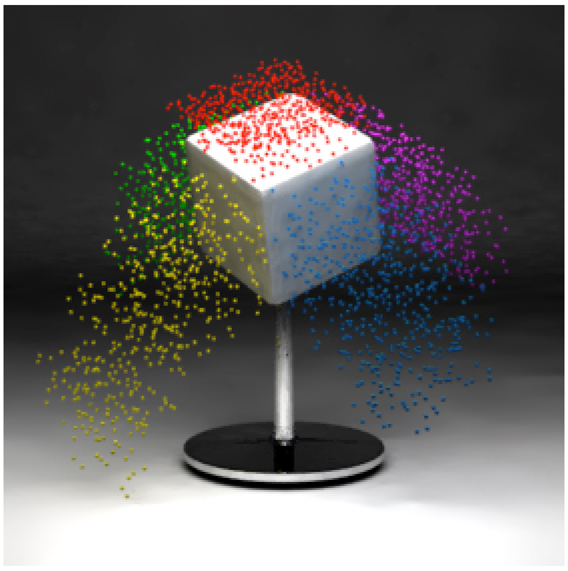
2.4.2 Particle Systems

For objects that do not have defined surfaces, such as clouds, fire and water, and as such can't be modelled with surface based modelling, a more specialised form of rendering is required. The particle system was first proposed by William T Reeves in 1983 [8] to render a sequence of a bomb exploding in the movie *Star Trek II: The Wrath of Khan,* and models these 'fuzzy' objects as a cloud of particles, that can then be added to, change form, and eventually die. Particles are small, simple images or meshes that are displayed or moved in great numbers [9]. These properties are critical for generating the rain effect required, as the particles will start from a defined point at the top of the scene, then update as they fall through the scene, and then after they have fallen to the bottom of the scene then they are placed back at the top.

Particle systems are widely used in the game industry to simulate effects such as fire, sparks, explosions, fog, clouds, falling leaves and rain. They can also be used for more abstract effects, such as magic spells or trails to a players next objective, by having the particles fade quickly and then be re emitted at the source.

A particle system being used to render the effects of a spell in a Harry Potter video game {4}

In a typical implementation of a particle system, the position and direction of the particles in a 3D space are controlled by an emitter, which sets the point where the particles are generated from and which direction they move in. The emitter will define a shape, such as a point, line or plane, to represent the area in which new particles can be generated. The emitter can even take the form of a mesh to represent a particle emitting volume, as shown in this example of a cube.



Using a cube as a particle emitter {5}

The emitter has a set of parameters attached to it that govern the behaviour of the particles. In Reeves' original system, each particle had the following values:

- Position
- Velocity
- Colour
- Lifetime
- Age
- Shape
- Size
- Transparency

Often these values are themselves fuzzy- the artist would specify a value for each parameter and then a degree of randomness that is permitted on either side. Depending on the mesh object being used, the initial velocity of the particles could be set to be normal to the faces of the object to give the impression the particles are spraying from each face.

Updating particles

Firstly, the number of particles that need to be added are calculated based on how many particles are generated per time interval, and they are spawned in 3D space based on the position of the emitter and the area specified. Each particle has its parameters set according to the values in the emitter. Then, all the particles are checked to see if their lifetime has been exceeded, and removed from the system if this is the case. If they are still alive, then their position needs to be updated- this could just be a case of translating the particle, but in a realistic game, physics calculations would need to be performed, accounting for friction, wind and gravity. Collision detection between the particles and objects in the scene would then be performed, to make the particles bounce off and otherwise interact with the environment. Collisions between particles are difficult to implement, since particles have no volume, and the process is computationally expensive. Therefore, this feature is rarely implemented.

After the position of the particle is updated, then each particle is rendered to a small graphical primitive, usually a quad [10]. In Reeves' original model each particle was displayed as a point light source. Therefore, each particle is displayed, and there are no hidden surfaces that need to be determined. If a particle is behind another one, the colours are added according to the transparency values.

Static Particles

If the life cycle of each particle is rendered simultaneously, showing the particle's trajectory rather than a point, then the resulting strands can be used to simulate hair, fur, grass etc. The strands can be governed by the same parameters as an animated particle system. Additionally, the thickness of each strand can be varied, having different thicknesses along the strand.

A cube rendered using static particles {6}

Optimisation of Particle Systems

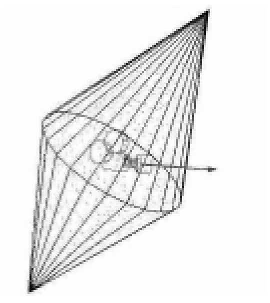
Since particle systems involve overlapping primitives that are translucent, when the pixels are written the z-buffer is not updated and pixels are drawn to multiple times (overdrawing) [11]. This leads to increased fillrate (how many pixels can be rendered to per second) and bandwidth (data transfer on the GPU). There are many ways of reducing this:

- Cap total particles- count how many particles have been rendered and stop emitting/drawing particles when a set limit is reached
- Use opaque particles- disabling alpha blending, so the z-buffer is written to, z-culling and depth testing can take place, which in turn leads to little overdraw
- Reduce state changes- share shaders between particles to reduce calls to shaders
- Reduce sorting- if the particles can be drawn in any order, then they don't need to be sorted on depth
- Free list- by implementing 2 lists of particles as opposed to one, with one list being drawn and the other storing the data from expired particles, the number of calls to new and delete are reduced and memory bandwidth is saved. The free list can also be dynamically reduced over time, since the slower the launch rate of an emitter the smaller the free list would need to be- if it were too large there would be unused particles unnecessarily taking up memory.

Alternatives to Particle Systems

Textured Cone

A system of rendering rain and snow using a textured double cone was developed by Wade and Wang for the game *Microsoft Flight Simulator 2004* [12]. This works by placing 2 cones around the camera.

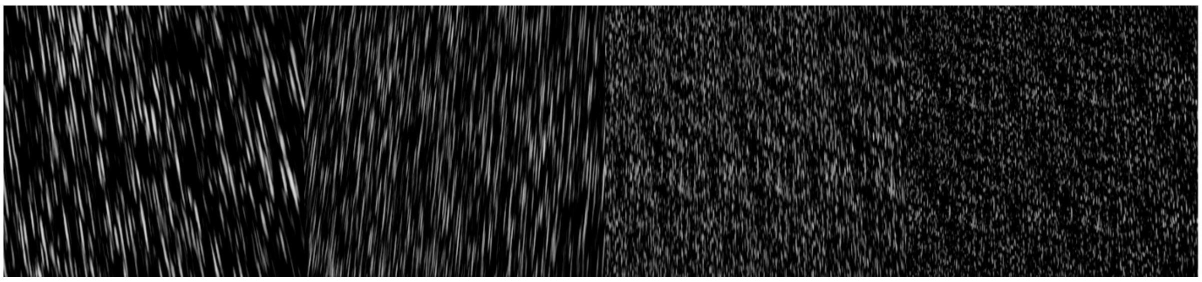


A precipitation texture is tiled to cover the 2 cones, and then blended together with a vertex shader. Then, the cones are tilted to account for camera movement, and the textures are elongated and scrolled to simulate motion blur and gravity respectively. This method is faster than a particle system, but doesn't allow for interaction between the precipitation and the landscape. Furthermore, a new texture would have to be loaded for every different form of precipitation that needed to be rendered using this system.

Case study: Remember Me

Remember Me concept art {7}

The rain effects in *Remember Me* (Capcom/Dontnod entertainment,2013) are interesting because unlike most 3d games that use a particle system, this game uses a texture based solution [13]. This works by mapping multiple layers of a rain texture onto a cone, rendered at the camera's origin. Then the different layers can be transformed and rotated at different speeds and positions to simulate movement.



4 layers of rain texturing used in Remember Me {8}

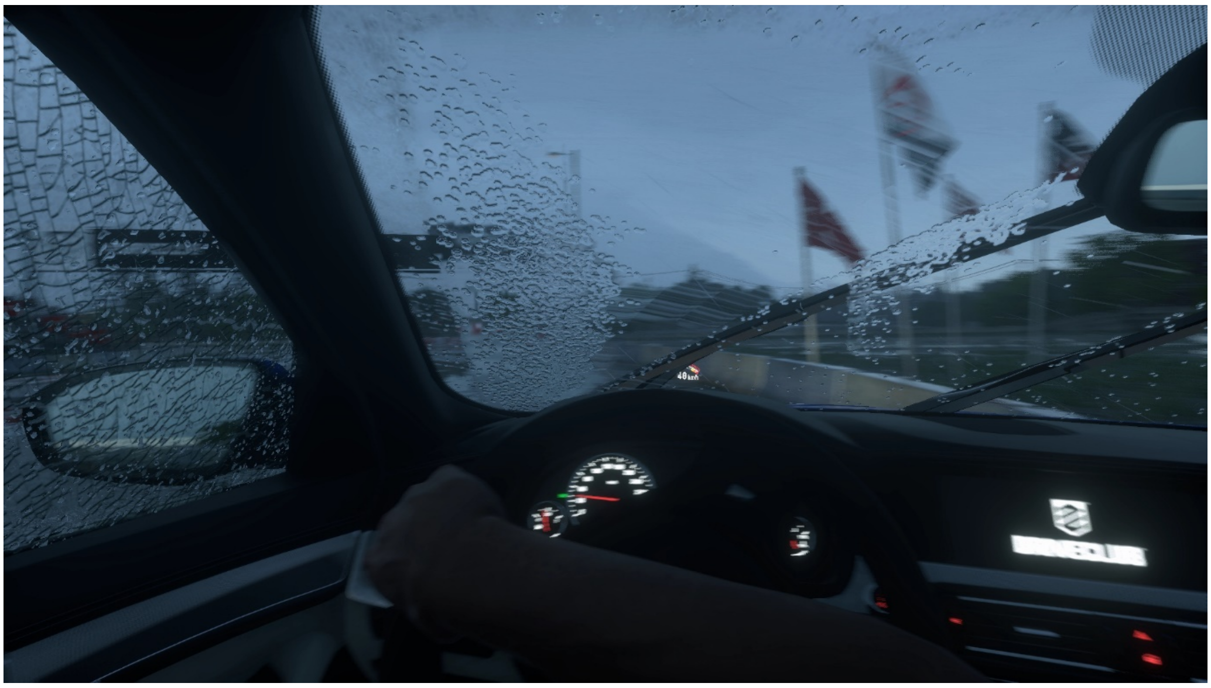
Each of the 4 layers uses a progressively larger scale factor to increase the intensity of the rain. Furthermore, each layer can have different effects applied to it, like raindrops falling down the camera and falling droplets on walls.

Post Processing

In the rainfall rendering portion of the series of graphical effects presented by Tatarchuck and Isidoro, the 2 authors combine multiple layers of falling raindrops in a single compositing pass [14].

The artist can specify the rain direction and speed, and from this the texture coordinates are computed. A parallax parameter is also supplied by an artist, which maps the depth range of the rain. Then, multiple layers of rain can be modelled in a single pass with one texture fetch. This method is efficient and looks good, but doesn't interact with the environment (later in this paper particle systems are introduced to model collisions), and doesn't allow manipulation of the rain directly.

Case Study: DriveClub



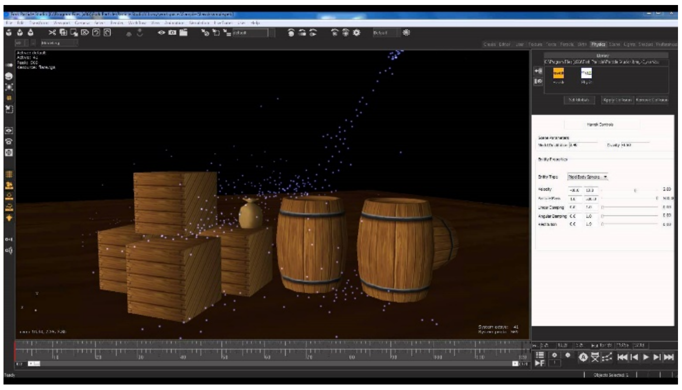
Screenshot from DriveClub {9}

It is widely agreed [15] that the most realistic looking rain in a modern game is in *Driveclub* (Sony/Evolution Studios,2014). The rain particles are each individually impacted by motion blur and properly lit by other car's headlights. Rain droplets dynamically fall onto cars, and then streak realistically according to the car's acceleration and are pushed around by the wiper blades, as shown above [16].

There is a lot of debate as to how the rain in *Driveclub* is rendered [17], but it's assumed to be a combination of looping animations (for the side windows), and then a 2D water simulation for the windscreen, to account for variables such as how fast the car is accelerating, its angle of turning, and the position of the windscreen wipers. Once the position of the particles has been calculated, then they can be grouped into a metaball(n-dimensional object) and have a texture applied.

Industry Particle System Implementations

- Havok

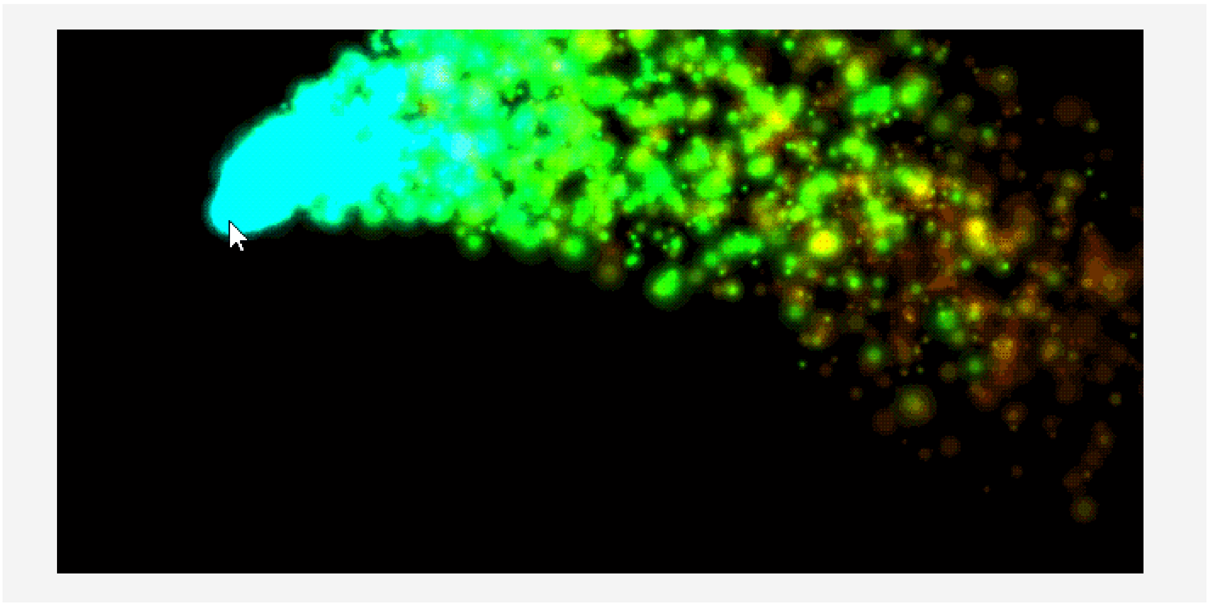


The Havok FX API provides a flexible effects solution for high performance particle physics and debris simulation [18]. Havok is also used in Valve's source engine [19].

- PhysX/Nvidia

Nvidia FleX (previously PhysX particles) [20] is an API used in Unreal Engine 4. It is a unified particle solver, supporting collisions between rigid and soft bodies. It is used for cloth, fluid, particles, ropes etc. [21]

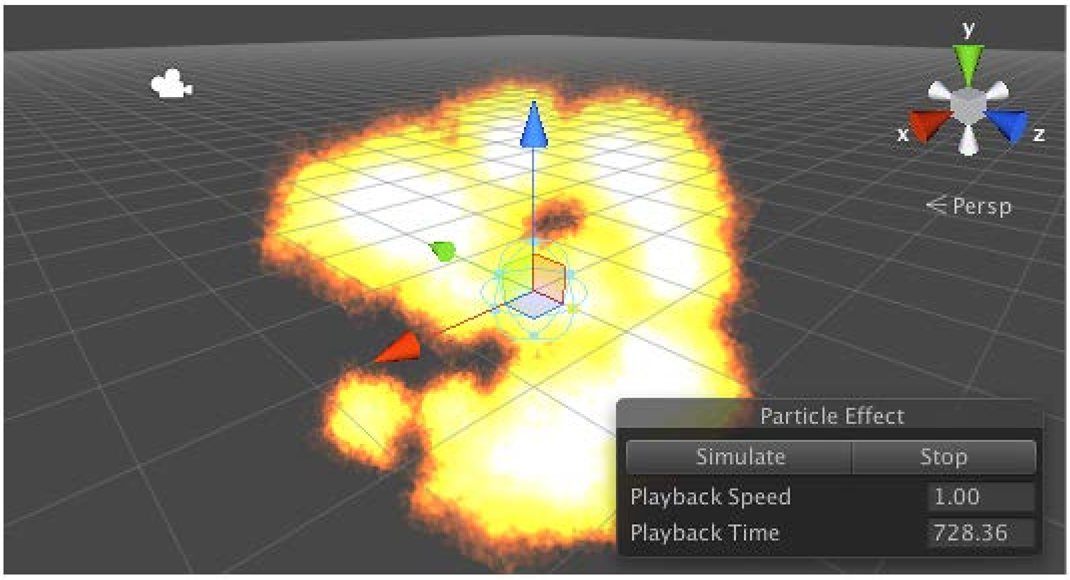
- GameMaker Studio



Particles in Gamemaker Studio {10}

GameMaker Studio offers a versatile 2d Particle system [22] that is used in games like *Undertale* and *Hotline Miami*

- Unity



Playback Speed 1.00
Playback Time 728.36

Particle system in Unity {11}

Unity's inbuilt particle system, called Shuriken, allows particle systems to be placed into a game scene and then modified [23]. Unity is used mostly in mobile games.

PART 3: DESIGN/IMPLEMENTATION

The three main components of the test scene are the terrain generated by a height map and index buffering, the rain itself generated using a particle system, and the user interaction with the scene and its elements.

3.1 The Decision to use OpenGL

Of the 2 options detailed in the background research section, I decided to choose OpenGL. The benefits of OpenGL are detailed on the official OpenGL website [24], but the ones pertinent to this project can be summarised as follows:

- The OpenGL command library is logical and English-like, which means it is easy to understand and allows non- users to comprehend what a program is doing
- OpenGL is a platform independent API, so code written with it can be ported between systems with little fuss. It produces consistent results on all OpenGL API compliant hardware, regardless of operating system or windowing system.
- OpenGL has been available for over 25 years, so there is a great amount of documentation available for it. There are many books published and a large amount of sample code available online.
- OpenGL is a very stable API, since backwards compatibility requirements ensure that code written in OpenGL does not become obsolete.

The major contributing factor in my decision to use OpenGL however, is the existence of the nclgl framework.

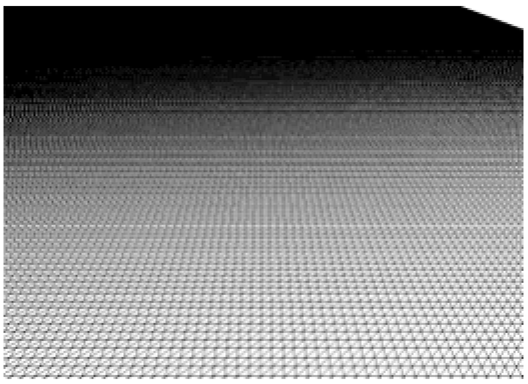
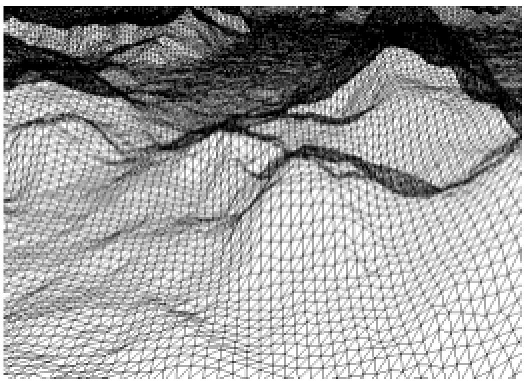
3.2 NCLGL

NCLGL is a framework that uses OpenGL, as well as SOIL (Simple OpenGL Image Library) [25] and GLEW (GL Extension Wrangler) [26]. It was written by Richard Davison for the use of students in Newcastle University. The nclgl framework provides a renderer class that encapsulates the OpenGL API, and a mesh class that encapsulates the vertex data the Renderer outputs to the screen, writing data to Vertex Buffer Objects (VBO) to send to the graphics card.

I already have experience of using this framework in modules I have taken in 3rd year, ‘Graphics for Games’ [27], ‘Gaming Simulations’ [28] and ‘Computer Games Development’ [29]. With this previous experience I had gained, I was confident that NCLGL had the required functionality to adequately meet the objectives of this project. Because I was using this framework, the decision was also made for me to code in c++, which was a programming language I was already familiar with.

3.3 3D Landscape

To visualise the rain particles in a context in which they would be featured in a game, the particles in my test scene will fall over a section of a basic landscape. Using a heightmap will provide a realistic environment that is efficient to render, and can be used to demonstrate particle interactions.

A 2D grid of triangles can be generated with a simple algorithm(left), and then using a map of different height values, which has been constructed by the TerraGen heightmap generation tool [30], the triangles can be set to different heights and a realistic looking 3D landscape can be formed. Index buffering (See 2.4.1.1) will be used to save memory.

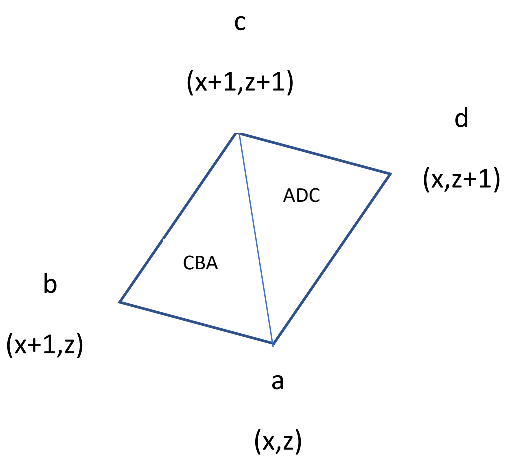
The heightmap class is a subclass of the nclgl mesh class, that provides a specific ordering of the vertex data for reducing memory. Since all the methods required are already in mesh, all the functions are handled in the constructor. As decided previously the height values will be generated in a RAW file by TerraGen, and takes the form of an array of 257*257 unsigned chars. This leads to 393,216 indices but only 66,049 unique vertices. Therefore, by using a heightmap, a reduction of 83% in memory usage is achieved. Both the vertex and index data has to be buffered to the GPU, by generating a VBO, binding it to the Vertex Array Object (VAO) and then buffering the data.

The file is passed in as a string into the constructor, and read into a temporary buffer of chars, and then used to populate the vertex array. The data is converted from a char to a 3D vector through nested for loops, then the data is deleted to save memory.

```
Int i =0
for x=0 to x= heightmap width -1
    for z=0 to heightmap depth -1
        int a= x*heightmap width + z
        int b= x+1 * heightmap width +z
        int c= x+1 * heightmap width + z+1
        int d= x*heightmap width + z+1

        //1st triangle
        indices[i]=c
        indices[i++]=b
        indices[i++]=a

        //2nd triangle
        indices[i++]=a
        indices[i++]=d
        indices[i++]=c
    end for z
end for x
```



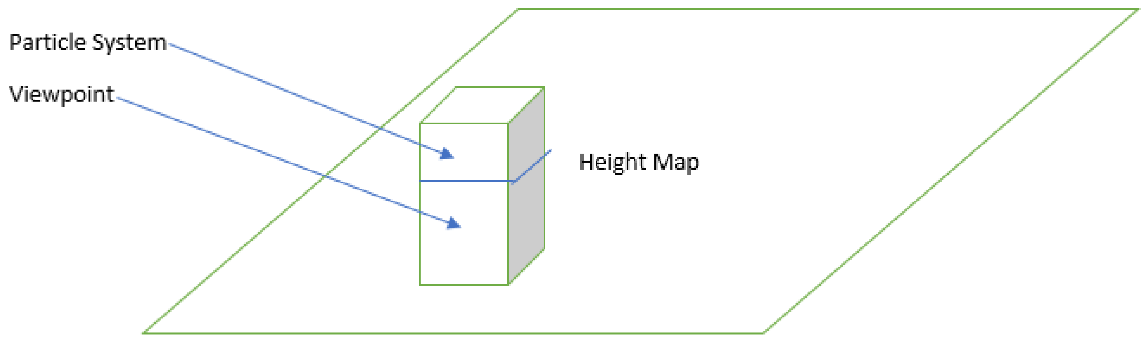
The above pseudocode shows how an index generation algorithm populates the indices array. For each square patch in the heightmap, the index data of 2 triangles are added to the array of indices. Vertex values a and c are being reused in each square, but instead of having to duplicate their vertex data, as would be the case were a heightmap not being used, a new index is written with the vertex value.

3.4 The Particle System

As detailed previously the decision was made to implement the rain effect as a particle system, in lieu of any acceptable alternatives. To satisfy the main project objectives, the particle system must have the following features:

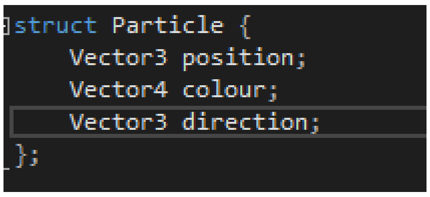
1. The rain must be able to be turned on and off with a key press at the user's command
2. The rain must be drawn as points in 3D space, which can then be manipulated by a geometry shader and turned into quadrilaterals to change the size of the raindrops.
3. The number of particles must be set before the program has run and have memory correctly allocated
4. The particles must be drawn and updated to simulate falling rain
5. The particles must occupy the area around the camera and move in accordance with the player- it would be inefficient to emit particles across the whole scene
6. Upon exiting the program, the memory allocated to storing particle information must be deallocated

In every frame of the scene, the particle system will be updated. The position of the particles exists in a 3D volume, which is then translated in relation to the position of the camera. This gives the impression that the rain is falling overhead and as such rain is covering the whole scene, regardless of the user's viewpoint. Having the particles follow the camera is an optimisation to ensure the particles that are being rendered are used in the most efficient way possible, since particles that are too far away to be seen or judge the distance of are not processed.



A basic particle system implementation is provided as an extra framework tutorial, and it has since been modified for the purposes of this project. This class is also a subclass of Mesh, but unlike the HeightMap earlier overloads Mesh's update and draw methods with its own functionality.

3.4.1 Particle Structure



```
struct Particle {
    Vector3 position;
    Vector4 colour;
    Vector3 direction;
};
```

The particle struct stores basic information about each particle, its colour, position and direction. Then 2 vectors are made of pointers to particles, an active list of particles that are currently on screen, and a free list of spare particles. When a particle is deleted from one list, it is placed on the other one. When a new particle is required, the free list is checked before any new particles are created. This approach saves memory by reducing new and delete calls.

3.4.2 System Variables

The system variables represent aspects of the particle system that can be changed. The particle class contains methods to get and set the following:

- Particle Rate: how often the system emits new particles
- Particle Lifetime: how long before the particle is deleted
- Particle Size: how big each particle is- this data will be passed to a geometry shader (Section 3.5.2.3)
- Particle Variance: A number between 0 and 1, representing how far the angle of each particle launched can vary from the starting point. Since this project simulates falling rain, a variance of 1 will be used, but if a direct stream of particles from the source of the emitter was required this value would be set to 0, negating later direction values.
- Particle Speed: linear velocity of the particles
- Launch Particles: how many particles the emitter launches each frame
- Particle Direction: the direction the particles launch in
- Particle Colour: colour of the particles

Whilst all these values will be changed later, either through update calls or directly by the user, they are set to defaults in the constructor. The constructor also loads the raindrop texture through the **SOIL_load_OGL_texture** function. There is also a variable for the maximum number of particles available in the system, which will be changed for testing purposes later. Finally, the constructor allocates the system and VBO memory, since in this implementation this will not change.

3.4.3 The Free List

To save memory bandwidth, the particle system employs a free list to recycle old particles and reduce the number of calls to new and delete. When a particle's lifetime is less than 0, instead of being deleted from the particle list it is instead pushed onto a free list before being erased from the active particle list. Like the particle list, this is a vector of pointers to particles, except this list isn't drawn. In the destructor of the particle system, each of the vectors are iterated through and the particles contained within are deleted.

3.4.4 Updating Particles

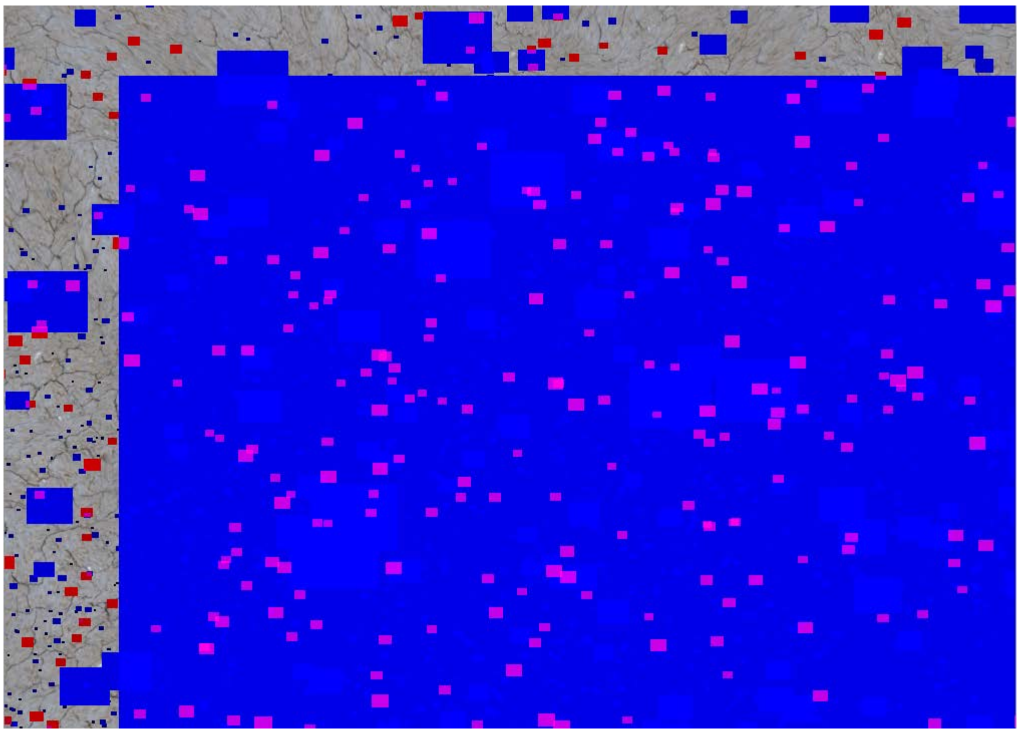
Like the method it overwrites, the *Update* function takes a float value of the amount of time passed since the last time update was called. This value is then subtracted from the particle rate to work out how many launch iterations are needed. Provided that the size of the particles list is not greater than the maximum number of particles allowed, the required number of particles are added to the active particles list, using the free list first. Once the active list has the correct number of particles in it, the vector is iterated over. Firstly, the particles 'life' is updated- this is stored in the alpha channel of the colour vector, so each particle becomes less opaque with time, fading out until it is completely transparent and 'dead'. If the alpha value is less than 0, the particle is erased from the active list and pushed back onto the free list. If the particle is still alive, then its position is updated by the particle speed multiplied by the particle direction. If the particle is deleted, then the iterator is not updated, otherwise particles would be skipped over.

3.4.5 Obtaining a Free Particle

The *GetFreeParticle* method returns a pointer to a free particle- either from the free list or by creating a new one. If the size of the free list is greater than 0, the particle at the back of the list is returned using the *pop_back* method. If not, a new particle is created. Then, this particle's information is either instantiated or reset- if the particle has been retrieved from the free list then it will still have the old values at the point of deletion. Of interest here is the particle's direction- the x and z values are randomised, but the y value must always be -1 since the system is simulating falling rain and the particles should move downwards. The direction vector needs to be normalised, and its position is set to zero- the position of the emitter.

3.4.6 Drawing Particles

The overloaded *Draw* method binds the colour and vertex data to the VAO, in a similar way to the draw method of the mesh class. It uses the OpenGL function **glBufferSubData,** which differs from **glBufferData** by not allocating any new memory, just reusing the memory allocated in the constructor. This function also enables additive blending by use of the keyword **GL_BLEND.** This means that overlapping particles will have their colours added together.



Once the VAO has all the data it needs, the function **glDrawArrays** is used to draw all of the vertices. The function takes in the type of primitive to draw (**GL_POINTS)**, and the range of values to draw, in this case the size of the active list of particles.

3.5 Shaders

Shaders are short programs that run on the graphics card and turn the vertex data passed into the VBOs into final image data. Shaders are written in the GLSL language, which must be complied and then attached to a shader object, which is accomplished with the shader class.

3.5.1 Shader Class

The constructor of the shader class takes in 3 parameters for the 3 different types of shader. The files are read in using a standard input stream and then compiled using the functions **glShaderSource** and **glCompileShader**. The shaders are then attached to the program using **glAttachShader**.

3.5.2 Shader Types

There are 3 different shader types. In this project, the height map uses a vertex shader and a fragment shader. The particle shader has all 3- the vertex and fragment shaders as well as a geometry shader.

3.5.2.1 Vertex Shader

Vertex shaders take in the vertices from the mesh's VBO and apply transformation matrices to them. The shader uses a model matrix, a view matrix, a projection matrix, and in the case of the heightmap, a texture matrix.

- Model Matrix

  The model matrix transforms a vertex from a position in local space (the coordinates loaded from the VBO) to global space (where objects are in relation to each other) using rotation, translation and scale matrices

- View Matrix
  The view matrix is essentially the model matrix for the viewpoint- it uses translation, rotation and scale matrices to control the view. In this project, the view matrix is built by the camera class.
- Projection Matrix

  A projection matrix transforms the vertices from global space to clip space- which constrains the vertices between 6 planes (left, right, top, bottom, near and far, collectively called a frustum), clipping and culling the primitives as necessary. It moves the vertices from the world origin to being relative to the camera position.

- Texture Matrix
  The texture matrix can translate, rotate and scale the texture co-ordinates

The model, view and projection matrices are multiplied together to form one matrix representing the concatenated result of all the transformations applied to a vertex. This is then multiplied by the vertex position and set using **gl_Position**. For the particle shader, the vertex colour is passed straight through to the fragment shader. The vertex shader for the heightmap doesn't have vertex colour data, it has texture coordinates that are multiplied by the texture matrix and then passed into the fragment shader.

3.5.2.2 Fragment Shader

Fragment shaders take data from the vertex shader and output fragments- colour data written to the colour buffer. If the object has texture data, the fragments are coloured with the GLSL **texture** method, which takes a texture sampler and texture coordinates, and returns the colour of the texture. This is then set with the **gl_FragColor** function. If the object has colour data, as with the particle system, then this is multiplied and output too.

3.5.2.3 Geometry Shader

The geometry shader is an optional stage in the rendering pipeline, and is placed in between the vertex and fragment shaders in the shader calling hierarchy. It takes the primitive from the vertex shader and then modifies it. In the case of this particle system, points (a single vertex) are taken in and converted to quads (4 vertices). This approach of expanding a point into a quad using the shader saves memory, since the vertex is always being updated by a set amount and it would be inefficient to have the particle emitter emitting quads

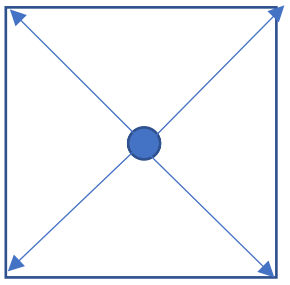
```
//Top Right
gl_Position = gl_in[ i ].gl_Position;
gl_Position.x += particleSize;
gl_Position.y += particleSize;
OUT.texCoord = vec2(1,0);
EmitVertex();

//Top Left
gl_Position = gl_in[ i ].gl_Position;
gl_Position.x -= particleSize;
gl_Position.y += particleSize;
OUT.texCoord = vec2(0,0);
EmitVertex();

//bottom right
gl_Position = gl_in[ i ].gl_Position;
gl_Position.x += particleSize;
gl_Position.y -= particleSize;
OUT.texCoord = vec2(1,1);
EmitVertex();

//bottom Left
gl_Position = gl_in[ i ].gl_Position;
gl_Position.x -= particleSize;
gl_Position.y -= particleSize;
OUT.texCoord = vec2(0,1);
EmitVertex();


EndPrimitive();
```



For each point, 4 vertices will be emitted. The x and y coordinates of each point are translated by the particle size variable from the particle system constructor (Section 3.4.2), and the texture coordinates are set accordingly. *EmitVertex* sends the vertex for rasterization, while *EndPrimitive* tells GLSL the primitive is complete.

3.6 Rendering the scene

The renderer class handles rendering all the elements to the screen. The constructor of the renderer creates a new instance of a heightmap, particle system and a camera, and loads the heightmap and particle shaders. Each frame the renderer draws the heightmap and the particles with the *RenderScene* function. Drawing the heightmap is a similar process to drawing the particle system (Section 3.4.6), but since it contains both index and vertex data the function **glDrawElements** is used. Then the *UpdateScene* function is called, which updates the particle system (See section 3.4.3) and camera. The heightmap does not need updating, since once its drawn it doesn't change.

3.7 User Interaction

A user must be able to interact with the elements in the test scene, through key presses and mouse movements.

General interaction with application

- The user must be able to look and move around the test scene, changing the viewpoint using standard keyboard and mouse controls present in many games
- The user must be easily able to exit the application

Interaction with the particle system

The user must be able to:

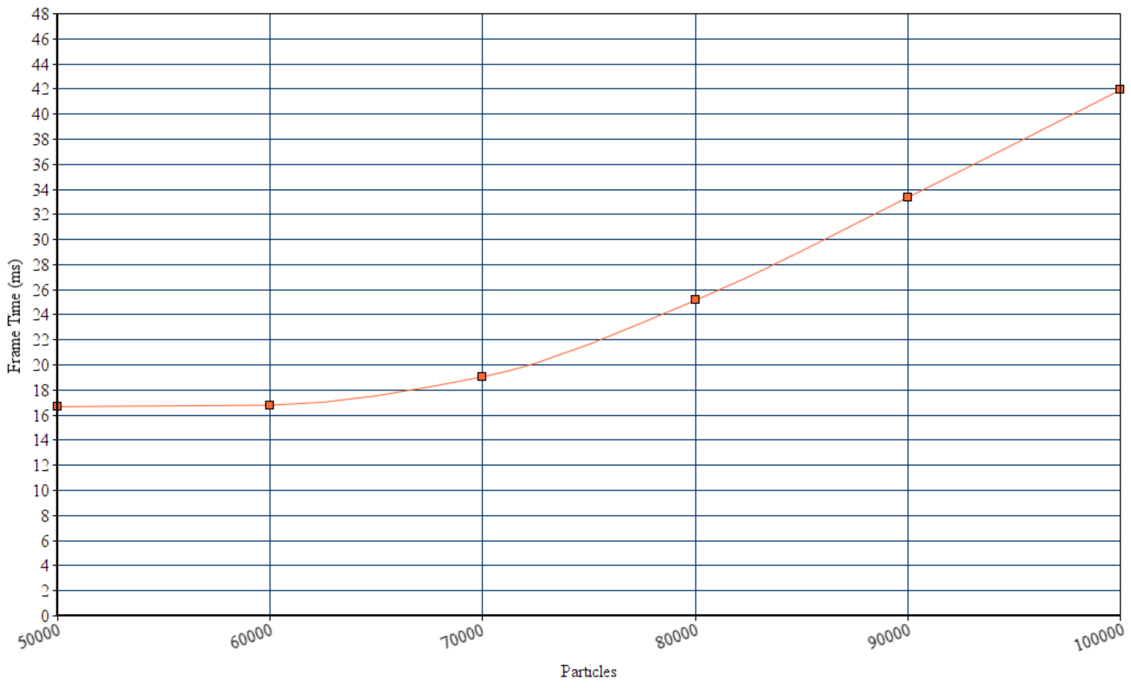
- Toggle the rain on and off
- Increase and decrease the size of the particles
- Increase and decrease the speed of the particles
- Change the particle colour (eg. Blue-rain, White-snow)

The keyboard class is used to register keyboard inputs and change the following variables

Key	Function
N	Increases the size of the particles
M	Decreases the size of the particles
P	Toggles the particles on and off
Z	Increases the speed of the particles
X	Decreases the speed of the particles
1	Change the particle colour to Blue
2	Change the particle colour to Red
3	Change the particle colour to Green
4	Change the particle colour to White
W	Moves the camera forward in the viewspace
A	Moves the camera left in the viewspace
S	Moves the camera backward in the viewspace
D	Moves the camera right in the viewspace

PART 4: TESTING/EVALUATION

4.1 Effect of Number of Particles on Frame Time

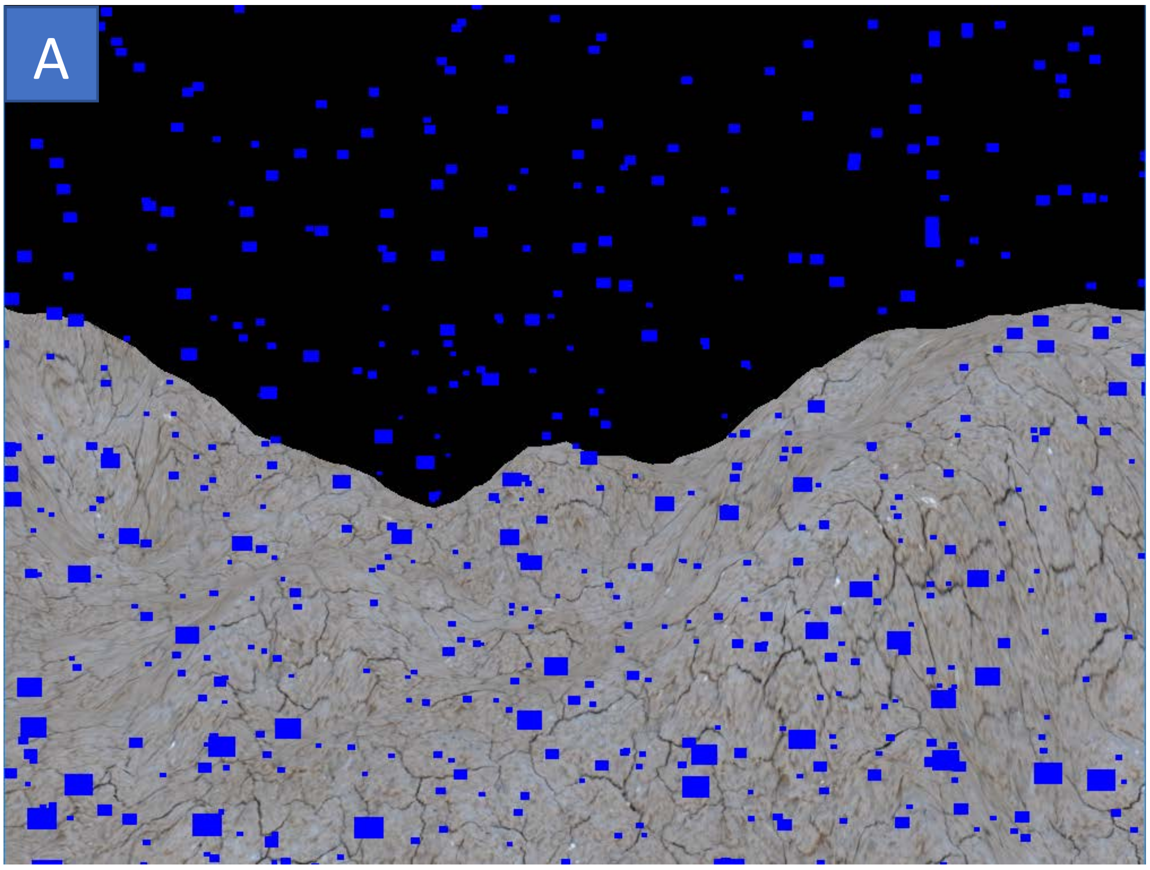


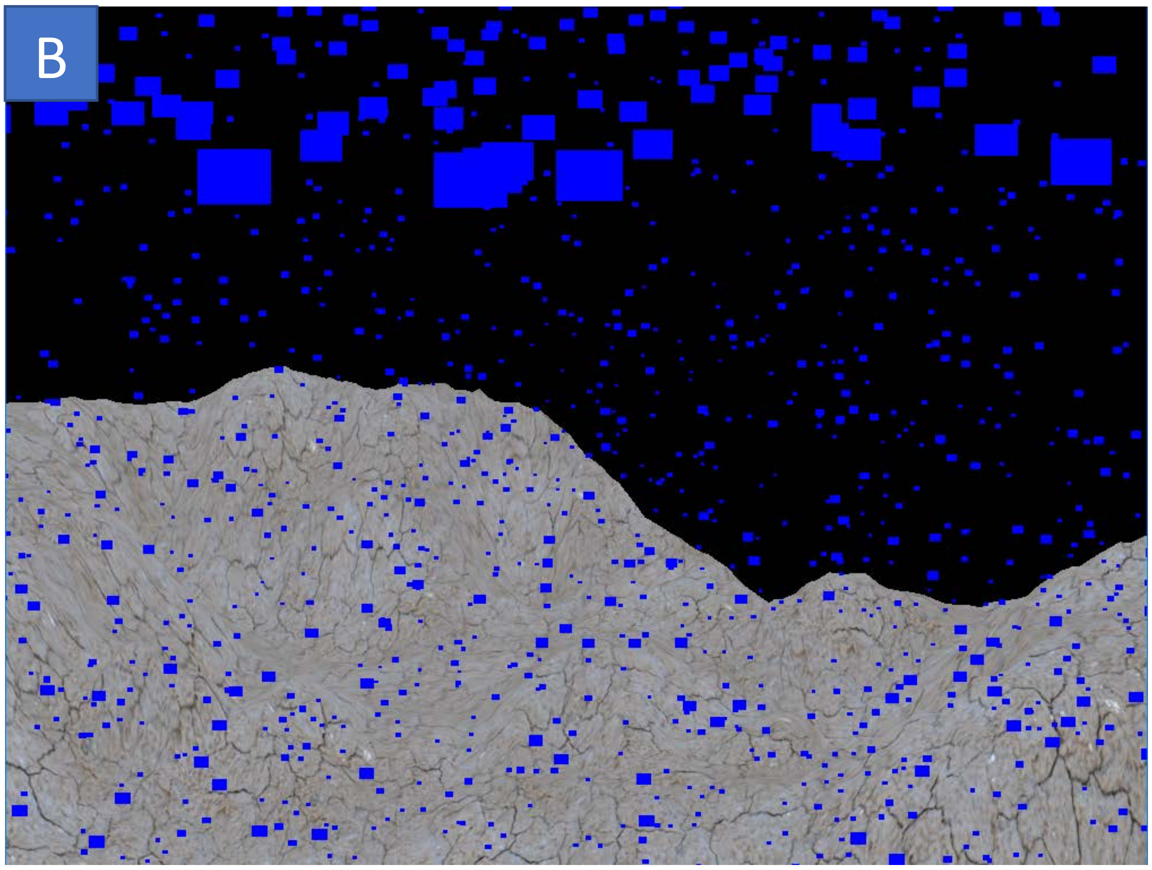
Graph showing the increase in frame time when the number of particles increases

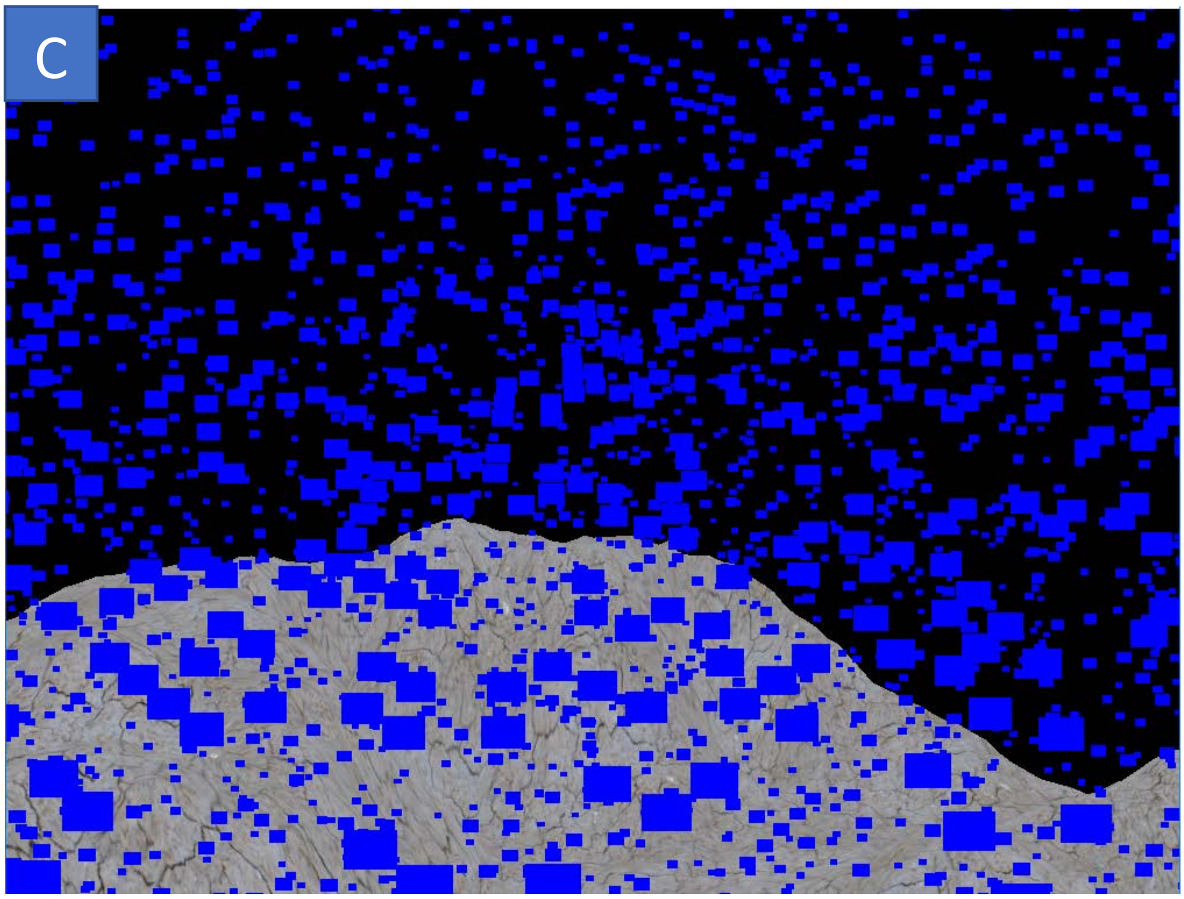
I tested the system by varying the number of particles using the max particles variable. The number of particles launched with each update was 1/10th of the max particles. The particles had a size of 1.5, a speed of 0.1, and a lifetime of 1000. These values were chosen as sensible defaults, making the particles big and slow enough to be able to be observed, and a with a long enough lifetime to occupy the entire viewspace before being recycled.

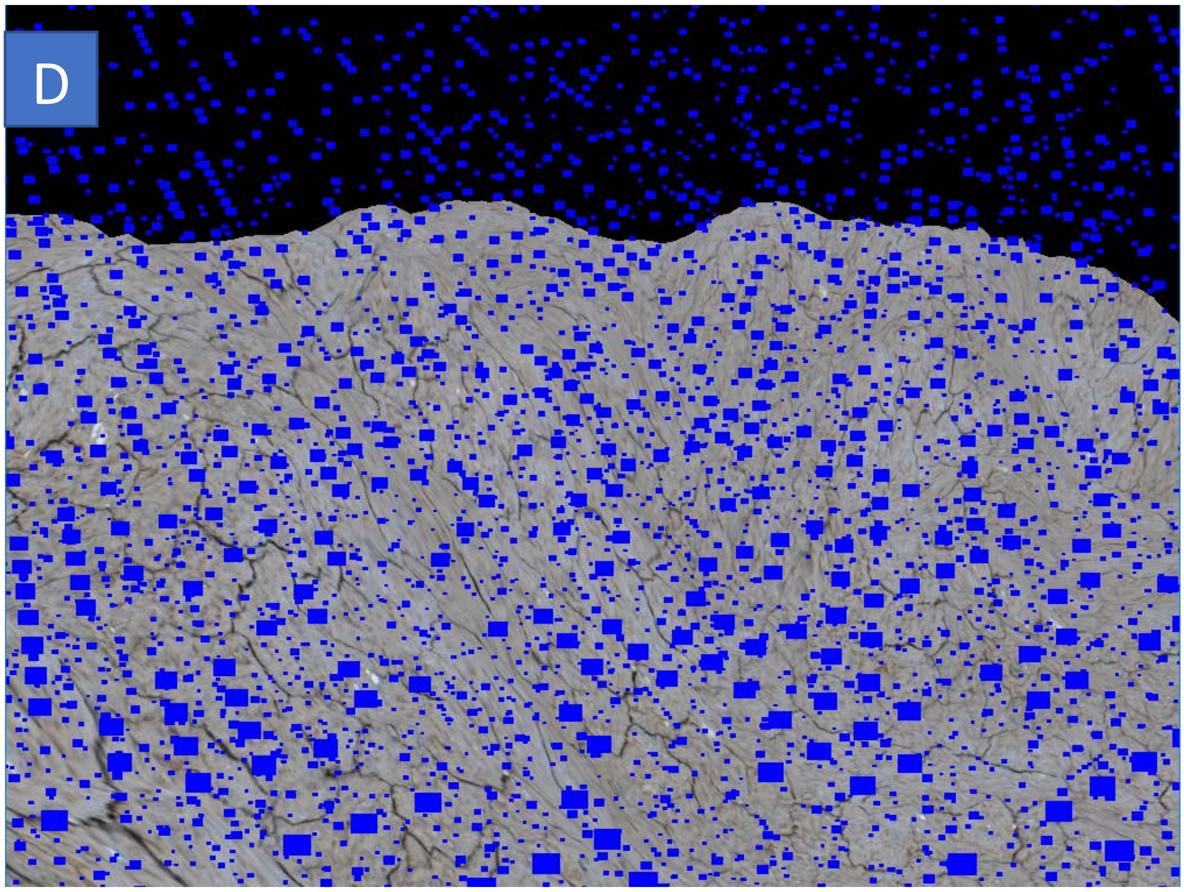
I measured the frame time using NVIDIA's NSight tool [31], taking each reading 3 times and calculating an average. As is expected, the frame time increases as the number of particles increase. Once the number of particles exceeds 60,000, the frame time begins to suffer, with each step of 10,000 increasing the frame time by approximately 8ms. This change appears to be constant, so despite the framerate becoming too erratic to obtain accurate results if the number of particles was over 100,000 on my computer, were I to continue testing on more powerful hardware I would expect this trend to continue.

4.2 Visual Inspection






C


D

A- 10,000 Particles
B- 25,000 Particles
C- 50,000 Particles
D- 75,000 Particles

I believe that a suitable number of particles in the simulation to be between 25,000 and 50,000. As discussed earlier (section 4.1) there is no loss in performance until the particle count exceeds 60,000, and the visual evidence above shows that the test scene with 75,000 particles looks both too busy and too ordered- since there is only a certain number of ways that 75,000 particles can be distributed they begin to form neat patches, which betrays the randomness a convincing implementation of rain needs. Conversely, the scene with 10,000 particles looks too sparse- this may be a suitable number if the rain were to be viewed from a distance, or the particle effect were used to model another effect such as sparks or an explosion, but for rain directly tied to a player's position in the world, I feel it is not significant enough to be effective. Test scene B is probably the best of the 4 visually, the rain falls in a substantial area, enough to have particles of varying sizes visible to the player, whilst the overall rain effect is intrusive enough to notify the player, but not bad enough to affect visibility or gameplay. Test scene C is very hectic, and is a convincing simulation of a heavy rainstorm. It could be used if a designer wanted to have a game where the rain had a tangible effect on gameplay, since having so many particles on screen at once leads to poor visibility and impedes a player's experience of the game world. However, this number of particles is at the upper limit of what is 'safe'- i.e. adding more sprites to the scene may start to produce slowdown, and the designer would need to consider this tradeoff and if they needed this many particles.
In my opinion, the ability to render more particles on screen at one time is not worth the drop in frame rate that occurs when the number of particles exceed 60,000. Indeed, the more particles added to the emitter the worse it begins to look, and the best results are achieved well before this drop off begins to occur. An optimal number of particles would be between 25,000 (for a light drizzle) and 50,000 (for heavy rain).

4.3 Evaluation

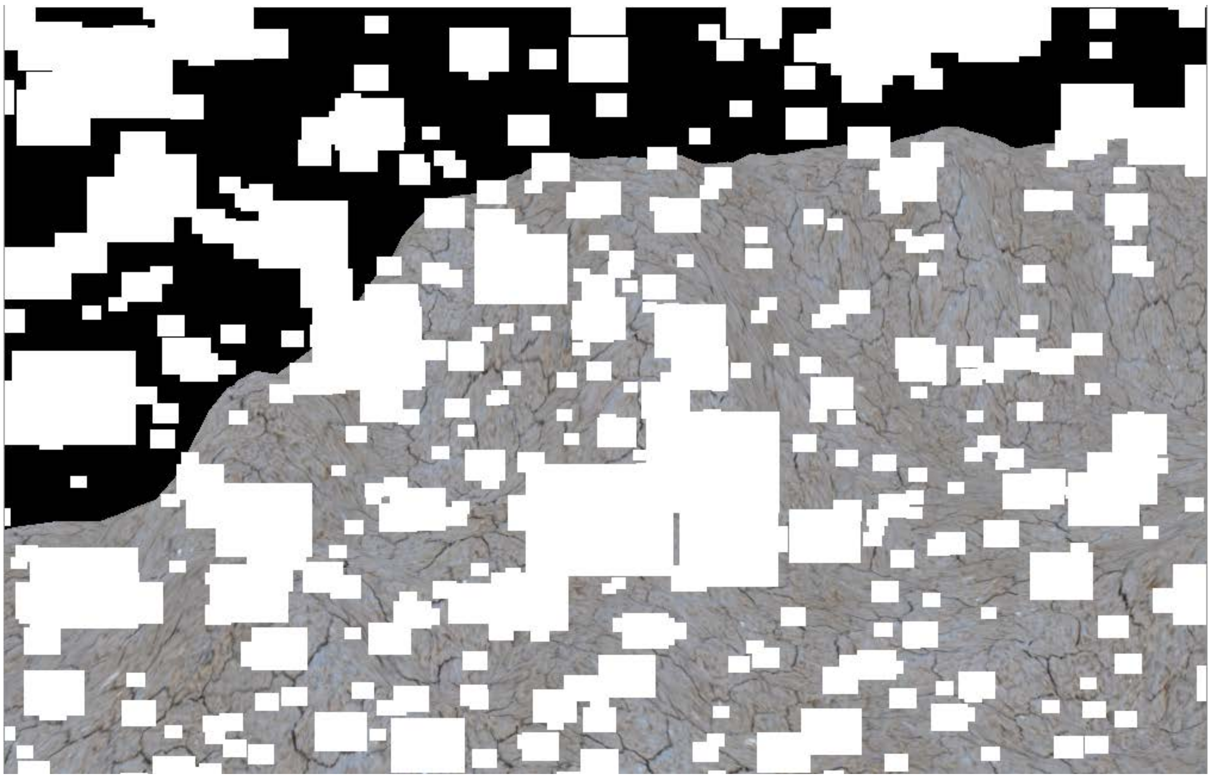
4.3.1 3D Landscape

There were no specific objectives relating to the 3D environment, since it only existed to provide a game context. As it is, the landscape is convincing representation of a rocky landscape, and is a successful implementation of a height map.
However, the height map has no impact otherwise, and doesn't affect the particles in any way. If I had additional time for this project, the main change I would make is adding functionality to detect collisions between the particles and the heightmap, either through individually checking each particle every frame, or a more elegant solution like a depth map [32]. Once collisions have been detected, effects can be applied. Different splash particles can be drawn upon a collision with the ground. An enhancement of this method would be to calculate the angle each particle struck the heightmap at and draw the splash accordingly, as well as having particles roll down hills or into dips and crevices to create puddles and

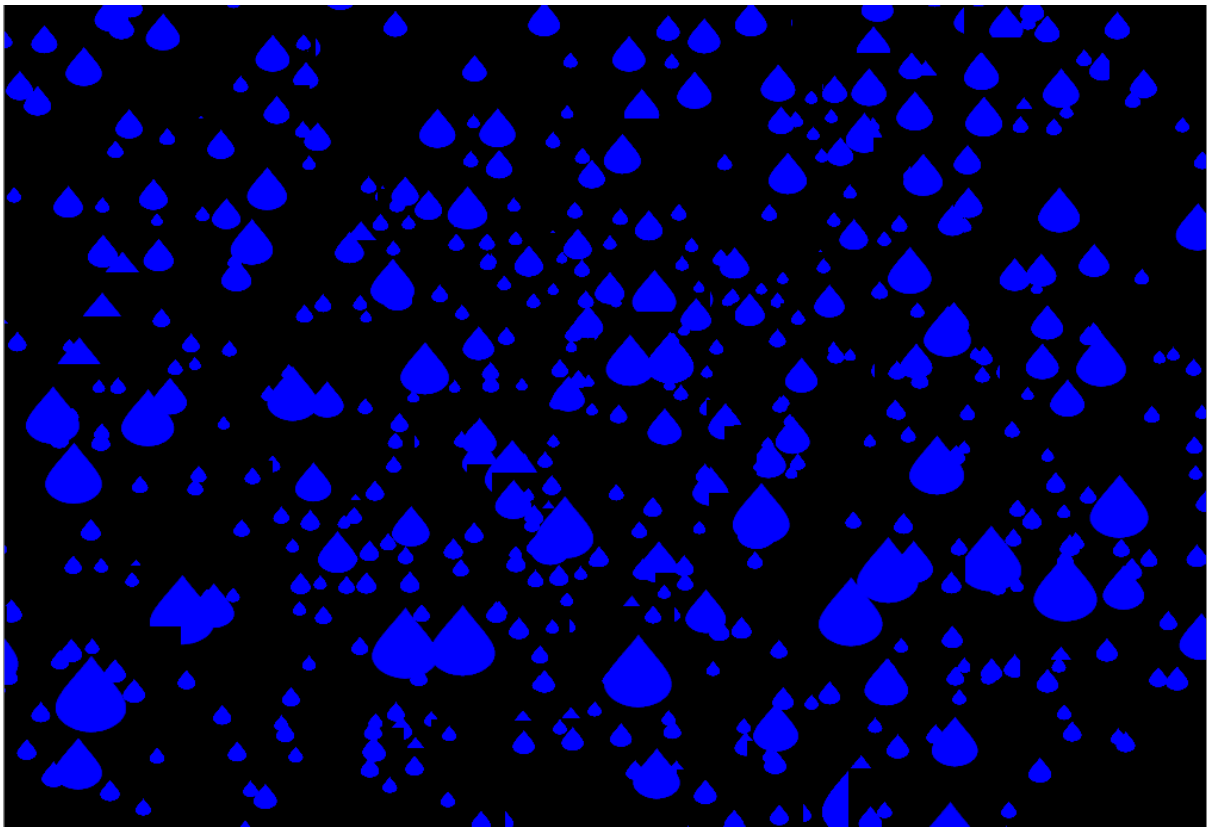
pools. Furthermore, a way of determining the impact velocity of the particle would need to be implemented to work out the impact size of the splash. Sound effects can be added upon collisions. Enhancements to the landscape's realism could be made by drawing additional objects in the scene such as rocks, or removing parts of the landscape to create overhangs, which would then react to being rained on in realistic ways.
The OpenGL lighting facilities could be used to implement a lighting system in the test scene, perhaps using an overhead light source to simulate the sun and the way it would cast shadows upon the scene. If collision detection was implemented, then it would be relatively easy to detect collision between the camera and the landscape, and so the user could be placed directly into the terrain. This would increase immersion, since the player would feel a part of the scene, a character in first person game, unlike how in the current build, where the user feels like they've enabled god mode, freely moving the camera around the scene. The player could then also be able to jump, which would lead to interesting situations such as jumping in puddles, but this moves out of the realm of particle systems and into fluid dynamics, another area of research. Since the landscape was not the focus of this project, it is very rudimentary and most of the development time was spent on the particle system.

4.3.2 The Particle System

The particle system fulfils all the requirements set out in Section 3.4. Additionally, more features have been implemented due to my pivot away from focussing on the particle system's interactions with the terrain to focussing on user interaction with the particle system. For example, by reducing the particle speed, increasing their size and changing the colour to white, a convincing snow implementation can be generated.



Whilst this system looks acceptable from a distance, due to the choice to render the particle system around the camera the system looks good in motion but suffers in screenshot, as shown throughout this document. Since all the particles are 2-dimensional squares being rendered in 3-dimensional space the lack of realism becomes apparent when the squares become too big or too slow. Improvements to the particle system would be to modify the geometry shader to change the particles to an ellipsoidal shape more representative of an actual raindrop. Additionally, I would buffer the data dynamically and increase the size of the arrays once the size of the particle list was big enough- this would eventually give a stable memory size. If there was a light source as suggested above, then the particles could consider refraction, reflection and other interactions with light- which would make the effect more akin to rain rather than generic particles. Another change that could be made would be to write a new graphics shader to change the particle points into lines, which would provide a different rain effect that would look more convincing as heavy rain from a distance.
I would also improve the texturing of the particles, considering the amount of clipping that occurs when a texture is applied.



This is due to the particles still being changed into quads by the geometry shader, despite the texture being teardrop shaped. In a real game development scenario, the art team would develop bespoke assets that would be passed to the designers, so the geometry shader would be written with the texture in mind.

4.3.3 User Interaction

The level of user interaction that has been implemented in the application satisfies the requirements set out in the design section. The user can fully explore the graphical environment using standard WASD keyboard and mouse controls. All manipulation of the particles themselves have been mapped to keyboard values (see table in section 3.7). This is an efficient implementation, since the values can be changed when the application is running and the results can be seen immediately. Currently, there are no checks in place to ensure the values for particle speed and size remain sensible, so the first improvement I would make is to introduce bound checks to ensure both values remain positive-at present negative values have no effect on the particle size- the vertices are flipped and since the particles are squares this has no visual change. However negative speed values would make the rain fall upwards, which would destroy any immersion that the application was simulating a rain scene. Also, additional colour effects could be added to simulate different particle based situations-for example yellow particles for sparks or grey, less opaque particles for smoke. Additional user interaction that could be added to the scene would be a proper graphical user interface. This would replace the arbitrary key bindings for changing particle properties and centralise all decisions into one main menu, that could either read in values from the console before the application started or render in the window, pausing the application then updating it based on these new values. As well as the existing size, speed and colour, other variables could be added to this initialisation menu such as maximum number of particles, particle lifetime and the number of particles released on each update.

PART 5: CONCLUSION

5.1 Fulfilment of Objectives

The aim of this project was split into 4 major objectives

"Investigate existing approaches to rendering rain in video games"

This objective was fulfilled through the research of existing rain rendering systems and specific video games as case studies. It was quickly discovered that particle systems comprise, or at least underpin most video game's efforts to render rain in real time, with only the cone based method used in flight simulators providing a completely different approach. However, this was considered an unsatisfactory approach to implement in this project and so the particle system was selected and taken forward to be implemented.

"Select an appropriate graphical framework to render a rain test scene"

This objective was fulfilled by researching DirectX and OpenGL, the two major graphical frameworks available. OpenGL was chosen due to the existence of the NCLGL framework and my familiarity with it.

"Implement the scene with the framework"

This objective was fulfilled with the application detailed in the implementation.

"Evaluate the performance of the scene when parameters are changed and effects applied"

This objective was fulfilled by altering the number of particles and measuring the effect it had on frame time, as detailed in the results section. The performance of the scene only reached unusable levels when the number of particles was over 100,000, showing that the system performed well.

The overall project aim was to implement and evaluate and efficient approach to rendering rain in real time. As explained in the evaluation chapter, the delivered application fulfils this aim, as well as providing additional functionality that allows the user to interact with the graphical environment and modify the particles within it. Additional functionalities and enhancements to the existing system that would increase the believability of the approach have been identified, but I believe my implementation represents a quality solution.

5.2 Personal Reflection

I found undertaking this project to be both interesting and rewarding. I have always been interested in the creative process behind video games, and learning about graphics during several modules of my third year developed the skills necessary to work on my dissertation. I feel I have developed my understanding and abilities in both research and programming.

This project is the largest piece of work I have attempted, and as such I was daunted by the amount of work needed to be completed throughout, especially in conjunction with other challenging pieces of coursework in other modules. The inclusion of several smaller deadlines leading up to the main dissertation hand in, as well as regular meetings with my dissertation supervisor, went some way into helping ensure I was always devoting enough time to working on this project. However, I still had problems with time management, and towards the end of the project I found myself with a great deal of work to do. Therefore, if I were to attempt another project on this scale I would start planning earlier than I did, which would avoid stress later and lead to a higher standard of work throughout.

I would also have spent more time researching which dissertation I wanted to do. Whilst I had a vague idea of wanting to do something in the realm of video games, the specific area was chosen for me by my project supervisor. Over time I developed a real interest in particle systems, but initially I struggled finding motivation to work on this project.

Even though this project involved juggling several tasks at once, I would often spend more time than necessary on difficult tasks, neglecting the rest of the work and making no overall progress. It would have been more effective to come back to the problems later and work on other, more achievable tasks in the meantime. I would also utilise my supervisor and the other staff more efficiently to correct problems as soon as they arose.

Initially, my intention was to focus on the interactions between the particles and landscape, but after too long trying and failing to detect collisions I decided to pivot my focus onto user interactions with the particle system. This decision was a major turning point in my project, but in hindsight took me too long to arrive at, costing me valuable development time.

5.3 Further Work

Whilst the delivered application is a system of particles, there is no reason why the program could not be expanded to a system of systems of particles- multiple particle systems that handle different types of particle with different properties. For example, if there was a system containing rain and sparks, then interactions between the two types could be modelled.

At present, the particles in the system undergo a basic translation every update call they are alive. The particle struct could be expanded into a particle class, and then perform complex physics calculations that account for gravity, wind etc. This would allow for different intensities of rain to be modelled more accurately, from light drizzle to heavy storms. During testing I discovered that changing physical properties of the particles had minimal to no impact on frame time, therefore adding variance to the particle data would increase the realism at little computational cost.

The free list system employed in the particle system is an effective optimisation, but could be further refined by reducing the free list as necessary, and storing the data as objects instead of pointers so only one VBO would need to be used.

The particles have no interaction with the landscape. Further work could calculate the angle the particles collide with the height map and have the particles impact and bounce off the terrain, perhaps puddling where appropriate.

Work could be done to ‘gamify’ the test scene, locking the viewpoint to the terrain so the user would experience the scene as a player and not an observer. A jump button could be added so players could jump in the puddles detailed above.

REFERENCES

[1] Barton, Matt. *How's the weather? Simulating Weather in Virtual Environments* Game Studies, September 2008

[2] http://projectbritain.com/climate.html

[3] http://www.web3d.org/x3d/what-x3d/

[4] https://developer.apple.com/metal/

[5] https://directx.en.softonic.com/

[6] https://www.opengl.org/

[7] https://gamedev.stackexchange.com/questions/106495/which-consoles-may-i-target-with-opengl

[8] Reeves, William T. *Particle Systems- a Technique for Modelling a Class of Fuzzy Objects* Computer Graphics, Volume 17, Number 3, July 1983

[9] Lander, Jeff *The Ocean Spray in Your Face* Game Developer, July 1998

[10] https://docs.unity3d.com/Manual/PartSysWhatIs.html

[11] Ericson, Christer *Optimising the Rendering of a Particle System* realtimecollisondetection.net, January 2009

[12] Wang N, Wade B *Rendering Falling Rain and Snow* ACM SIGGRAPH 2004, Page 14, 2004

[13] Seymour, Mike. *Game Environments- part B- Rain* fxguide.com, June 2013

[13] [14] Tatarchuk N, Isidoro J *Artist-Directable Real-Time Rain Rendering in City Environments* Eurographics Workshop on Natural Phenomena, 2006

[15] https://www.neogaf.com/threads/best-weather-effects-in-a-game.1240080/

[16] Linneman, John. *DriveClub revisited: is dynamic weather a game-changer?* Digital Foundry, December 2014

[17] https://www.reddit.com/r/GamePhysics/comments/2orew0/amazing_rain_physics_driveclub/

[18] https://www.havok.com/wp-content/uploads/2015/06/HavokFX2015.pdf

[19] https://developer.valvesoftware.com/wiki/VPhysics

[20] https://developer.nvidia.com/flex

[21] http://physxinfo.com/wiki/PhysX_FleX

[22] https://docs2.yoyogames.com/source/_build/3_scripting/4_gml_reference/drawing/particles/index.html

[23] https://docs.unity3d.com/Manual/PartSysUsage.html

[24] https://www.opengl.org/about/

[25] http://www.lonesock.net/soil.html

[26] http://glew.sourceforge.net/

[27] https://www.ncl.ac.uk/module-catalogue/module.php?code=CSC3223

[28] https://www.ncl.ac.uk/module-catalogue/module.php?code=CSC3222

[29] https://www.ncl.ac.uk/module-catalogue/module.php?code=CSC3224

[30] https://planetside.co.uk/

[31] https://developer.nvidia.com/nsight-visual-studio-edition

[32] http://3dstereophoto.blogspot.co.uk/p/software.html

IMAGE CREDITS

{1} https://www.fxguide.com/wp-content/uploads/2013/06/RM_Concept_Envirom.jpg

{2} https://www.fxguide.com/wp-content/uploads/2013/05/rainlayer.png

{3} https://cdn.gamer-network.net/2013/articles//a/1/7/2/5/8/3/1/dc_001.bmp.jpg/

{4} http://machinethink.net/images/3d-rendering/Geometry@2x.png

{5} http://www.hpetrov.com/gallery/yaga-witch-3d-character-wireframe.jpg

{6} http://www.opengl-tutorial.org/assets/images/tuto-9-vbo-indexing/indexing1.png

{7} https://cdn.videogamesblogger.com/wp-content/uploads/2010/06/harry-potter-and-the-deathly-hallows-part-1-game-screenshot.jpg

{8} https://upload.wikimedia.org/wikipedia/commons/1/1c/Particle_Emitter.jpg

{9} https://upload.wikimedia.org/wikipedia/commons/4/44/Strand_Emitter.jpg

{10} https://www.yoyogames.com/blog/444/get-started-with-particles-in-gamemaker-studio-2

{11} https://docs.unity3d.com/Manual/PartSysExplosion.html