James Burrell

Registration number 100263300

2022

Simulating a Waterfall

Supervised by Dr Stephen Laycock

University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

The aim of modelling complex fluid behaviour has consistently challenged the field of computer graphics. Nonetheless, successfully producing both physically and visually realistic fluid models has been influential in providing solutions to problems in a wide variety of fields. This paper documents the research and development of a Smoothed Particle System in order to simulate a waterfall that can be rendered in real time. The final application achieved a visually accurate representation of a waterfall in a typical 'plunge' topology that rendered in real time, with frame-rates ranging from 370 to 25 frames per second depending on particle count and quality.

Acknowledgements

I'd like to thank Dr Stephen Laycock for his knowledge, guidance and patience throughout this project and always providing encouragement during a difficult stage in my life. I'd also like to thank my family for their amazing support throughout my degree, and my advisor, Mrs Debbie Taylor, for her advice and care.

Contents

List of Figures

List of Tables

1 Introduction

The following is a report produced to accompany the software application I have developed during my final year. It contains details describing the context of the project, an overview and analysis of relevant sources and choices in planning and design. The report then moves on to explain key implementation points, experimental results and concludes with an evaluation of the work I have achieved and suggestions for how the application could be further developed.

1.1 Problem Description

Realistic simulation and rendering of fluid systems has been a topic of continual interest and development in the field of computer graphics. The movement of fluids from one area to another is both physically and visually complex and so presents a number of challenges when it comes to recreating them accurately. Computational fluid dynamics (CFD), a subcategory of fluid mechanics, provides ways of representing the movement of a liquid and its behaviour with solids mathematically.

The complexities of these mathematical representations are in many ways caused by the large array of physical variables that influence how a liquid behaves. Density, viscosity and compressibility of a liquid are some of these physical attributes. Environmental factors such as temperature, wind speed and its direction can also contribute massively to the way a liquid behaves. These factors and their high amounts of variation mean that there must often be a trade off between accuracy and performance when wanting to graphically represent the simulation in real time.

Nonetheless, accurately modelling the behaviour of fluids has a range of applications in a wide variety of fields such as medicine (Bluestein, 2017) and electrical energy generation (Sumner et al., 2010) and the development of immersive environments in film and video games (Gonzalez-Ochoa et al., 2012). This ubiquity causes the development of more accurate and efficient fluid simulation models to be increasingly worthwhile in the field of computer graphics, with the subject being a continual area for research and progression.

1.2 Project Aims

The aim of the project will be to develop a realistic representation of a waterfall that is rendered in real time. The specification for the project states the waterfall must occupy a three-dimensional environment that includes an area of moving water such as a river or stream that then drops to create the waterfall itself. The water should then enter a plunge pool at the bottom of the waterfall's structure. This structure should consist of surfaces such as rocks that the falling water can collide with in a realistic fashion. In addition, real time rendering of the scene must be produced at what is considered to be an acceptably smooth frame rate, approximately 20 to 60 frames per second.

Meeting the specification provides an array of challenges. For instance, the environment in which the waterfall is situated must be built so that the terrain appears natural but also offers appropriate points at which the waterfall can interact with in terms of collision. Implementing appropriate shaders and lighting models is crucial so that objects and water in the scene are rendered with the correct colour, transparency, reflections and shadow which will aid in producing a more realistic image. However, the most challenging aspect of the project will be to produce a fluid model that allows an accurate simulation of movement of water in the scene. The main cause of this challenge will likely be due to the use of relatively complex CFD equations alongside graphical rendering routines. Both must be balanced so that a realistic representation of a waterfall is produced and rendered with acceptable frame rate and memory performance.

2 Literature Review

With the context and aims of the project described, the next stage in development is to collate, read and review resources that are likely to provide useful information on the subject or provide details into example implementations. Initial reading into the topic showed a number of potential academic sources, which was then narrowed down to a total of 21 papers and journal articles.

The sources were ultimately selected in regards to their relevance to the topic, the credibility and reputation of the source they were taken from, and their ability to describe potentially complex implementation details to a reader with little prior knowledge of fluid simulation models and techniques. The following section of the report details themes, key topics, debates, possible gaps in research and general evaluation of

the papers themselves.

2.1 Navier-Stokes Equations

The Navier-Stokes equations (NSE) are a set of partial differential equations that can be used to describe the dynamic motion, i.e. flow of a fluid (Łukaszewicz and Kalita, 2016) and subsequently form the basis of almost all solutions relating to CFD problem solving. The Navier-Stokes equation for incompressible fluids is given as:

$$\frac{du}{dt} = -(u \cdot \nabla) \cdot u - \frac{1}{\rho}\nabla p + \gamma \nabla^2 u + f \tag{1}$$

The equation represents the change in motion of a infinitesimal point in the fluid and therefore can be used to represent the change in motion of a constituent particle. Dobek describes how the equation can be divided into four, with each part representing how a specific attribute of the fluid affects the fluid's velocity.

The first term, $(u \cdot \nabla) \cdot u$, describes how the divergence (the change in density of a moving fluid) influences its velocity. The second term, $\frac{1}{\rho}\nabla p$, represents the tendency of a fluid's particles to move away from areas of high pressure. The third term, $\gamma \nabla^2 u$, combines the viscosity of the fluid with the Laplacian operator, with the whole term representing how likely a particle is to influence neighbouring fluid particles to move. Finally, the fourth term, f includes any other forces that may be acting upon the fluid (Dobek, 2012).

A number of papers on the development of fluid modelling systems have been published that utilise NSE to produce realistic results. Lee and Han developed an interactive application that uses NSE to model the behaviour of shallow water. Whilst the implementation is realistic, they admit early in the paper that generating scenes using an NSE-based model is "very time consuming to compute for a large volume of water body" and so other methodologies for recreating large scale scenes of water movement, such as oceans, usually only focus on modelling the surface of the water without consideration of its volume (Lee and Han, 2010). However, Takahashi et al.'s paper on producing realistic animation of fluids uses NSE to model large volumes of water, but does not do so in real time (Takahashi et al., 2003). Potentially the results of Takahashi's system could have shown real time rendering capabilities if conducted on more powerful hardware similar to what was available to Lee and Han in 2010.

It is also worth noting that the NSE equations used in the previously mentioned studies assume that the fluid being modelled is incompressible. This means that modelling of spray, foam, mist and other visually impactful elements of water flow cannot be accurately performed using NSE equations, and so these diffused or aerated elements of water flow are more accurately modelled using a smooth particle hydrodynamics-based method (Losasso et al., 2008).

2.2 Smoothed Particle Hydrodynamics

Smoothed particle hydrodynamics (SPH) is a method of modelling an array of complex systems, including fluid dynamics, by representing the subject as a set of spherical particles (Monaghan, 2012). The method was developed by Gingold and Monaghan in 1977, in an influential study that has since been cited nearly 9000 times across various academic publication sources (Gingold and Monaghan, 1977). It has since been reviewed in a number of follow up papers (Lind et al., 2020; Monaghan, 1992, 2005).

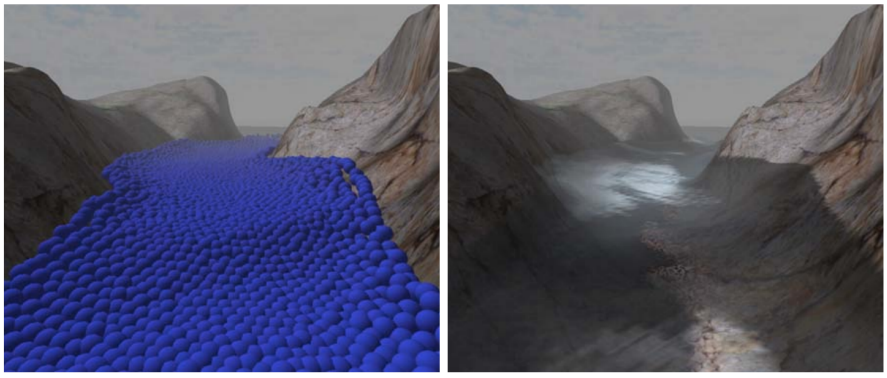

Figure 1: Using smoothed particles to represent a fluid (Bender et al., 2011).

The effectiveness and adaptability of the SPH method has been demonstrated in studies with applications ranging from astrophysics (as it was originally intended for when first developed) (Benz, 1988) to natural events such as lava flow (Husni et al., 2009). Another application by Suwardi et al. uses an SPH system to estimate electrical power generation from a waterfall, which in the context of this project is highly relevant (Suwardi et al., 2018).

The paper comments on the SPH method's merits, claiming the method "is easy to implement", and argues its adaptability by mentioning smoothed particle systems give

"the possibility of almost abundant water topology changes which could not be provided by most other methods". However, Suwardi et al.'s simulation only reproduces the movement of water in two dimensions and without the additional visual effects present in real waterfalls, and so this could be considered an area of improvement. Nonetheless, Suwardi et al.'s paper includes evaluation on the downsides of using a smoothed particle system to simulate water flow, with experimental results indicating that increasing the volume of water, and therefore particles needed to represent it, reduces the performance of the simulation. On the other hand, this appears to be a trend in all fluid simulations regardless of the underlying methodologies that are chosen.

2.3 Texture-based Rendering

Another trend that was identified during research was the use of texture-based methods for rendering complex fluid motion. These methods present an alternative to smooth particle systems and have been used effectively in producing realistic representations of varying types of water movement. In the infancy of computer graphics, most computer games (which is one of the primary applications for real-time rendering) modelled water by simply applying artist generated textures to a strictly planar surface (Johanson and Lejdfors, 2004). This method did not provide very realistic results, but this was true for all elements of a digitally rendered environment at the time.

Modern approaches include sampling 2D textures from pre-existing sources and then modifying them to conform to the desired water topology. An initial system was produced by Schödl et al. that replaced static textures with repeating dynamic 'video textures' consisting of multiple image frames (Schödl et al., 2000). The paper included testing on generating a scene that included a waterfall. The generated scene, whilst still not graphically high quality, produced a visualisation that was "still fairly convincing".

A more recent methodology proposed begins by sampling a series of 'basis' textures from a video sequence of a real waterfall (Guan et al., 2006). After this, a set of flow lines that represent the direction, shape and height of the desired waterfall flow path is created. These new textures are then rendered to produce the visualisation. Nonetheless, due to the fact that there is no underlying fluid model in this method, it lacks the means to dynamically model the motion of a fluid and its interactions with surrounding elements, such as collisions with rocks in this case.

Figure 2: Results of using a texture based approach to create a simulation of a waterfall (Guan et al., 2006).

2.4 Comparing Methodologies

It is clear that both smoothed particle and texture based systems for rendering complex fluid movement have their uses within the computer graphics world. For this reason, it is worth comparing the differences that their literature presents in order to better understand the contexts they are used in, the challenges associated with them, and their merits.

To begin with, most SPH methods rely almost exclusively on Navier-Stokes equations. This gives them the inherent ability to model any incompressible fluid accurately and as such means that SPH methods can be used for applications in a larger number of fields when compared to texture based methods. This ability to model measurable characteristics of a fluid accurately lends SPH methods to be almost exclusively used when it comes to CFD applications. On the other hand, rendering polygons with animated textures over terrain models can produce fairly realistic visualisations of waterfalls at much larger scales, unlike smooth particle systems, which when scaled up to larger scenes frequently encounter performance issues (Emilien et al., 2015). This makes texture based methods potentially more suitable for non-critical fluid modelling, such as large environments needed for some video games.

However, the papers identified during research that used texture based modelling pre-

sented certain pitfalls. For instance, dynamic texture sprite rendering involved projecting modified 2D textures onto a plane that was parallel to the view plane. This restricted the viewer to one position, and so 3D walkthroughs were not possible, making it unsuitable for many real time applications (Guan et al., 2006).

SPH methods can also present visual limitations, such as when they are not used in conjunction with appropriate surface smoothing techniques. Modelling fluids with particles often leaves surfaces looking 'blobby' or 'jelly-like' due to the sudden changes in curvature between particles. One might think to simply reduce the size of the particles to create a smoother appearing surface, but this then means that a greater number of particles are needed to represent the same volume of fluid, which comes at significant performance costs. For this reason surface smoothing techniques are a common addition to SPH systems to improve their visual realism. Methods such as screen space meshing developed in Müller et al.'s highly influential particle-based fluid simulations (Müller et al., 2003) or curvature-based screen space filtering by van der Laan et al. (van der Laan et al., 2009) have both generated smoothed, more realistic surfaces based on the position of the underlying particles.

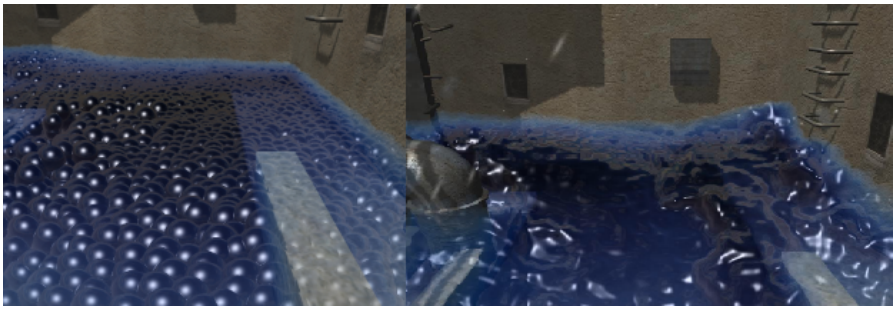

Figure 3: Example of an SPH model before and after surface smoothing (van der Laan et al., 2009).

Nonetheless, these techniques are not without their downsides; Müller et al's solution involves generating a high number of small polygons to form a surface mesh and so is computationally very demanding, and van der Laan et al.'s method has been described as "highly dependent on viewer distance" in order to appear realistic (Bagar et al., 2010).

2.5 Summary

A target area of my research that I expected to find more details on was the subject of scene lighting and shading. At first, it was assumed that documentation on the implementation of lighting models used for fluid simulations would be present in most if not all papers that were researched. Whilst some papers (Iglesias, 2004; Takahashi et al., 2003) did include details on how light should behave with water, many (Bender et al., 2011; Sakaguchi et al., 2007; Wong, 1994; Xiao et al., 2018) did not or sections where they were are brief. One possible explanation for this could be that methods for lighting scenes containing clear fluids, such as ray tracing, are seen as standard and so their use is not worth explicitly mentioning.

Nonetheless, the selection of papers gathered during preliminary research has provided insight into an array of key topics, the most common being Navier-Stokes equations for fluid dynamics, Smoothed Particle systems and Texture-based Rendering. All papers referenced in this literature review appeared to contain very few to no identifiable biases toward any particular methodologies without providing sufficient experimental evidence, whether that be from other credible papers or self-produced experiments. However, this should be the expected from scientific papers that have been taken from academic sources.

3 Design

With initial research and literature review completed, the next section of the report documents further research, planning and design elements of the projects. Also discussed are the reasoning behind the choice in methodologies, resources and tools.

3.1 Methodology Choices

Following the trends identified during research, it was decided that a system based on Smoothed Particle Hydrodynamics would be the most appropriate for the development of the waterfall simulation. An SPH based system has been chosen due to five main reasons, which are described below:

- **Iterative**; the method appears well suited for being implemented in an incremental approach.

- **Adaptability**; SPH systems provides the ability to develop a system that can be easily varied to suit different requirements, making it more likely to have potential use for other tasks.
- **Visualisation**; the method allows preliminary models of the system to be visually represented at an early stage, meaning both increases in functionality and potential faults can be identified more easily.
- **Documentation**; SPH is a popular methodology for the context, and so access to relevant literature should ease development.
- **Experience**; previous work with computer graphics lends itself to working with object movement i.e. particle motion

For these reasons, developing a simulation through an SPH based method seems more desirable than a texture based method. The SPH method should provide an overall more flexible solution and better end results which meet the specification described in Section 1.2.

3.2 Resources

The following describes the choices in tools that will be used to develop the application for simulating a waterfall. Many of the choices have been made in regards to previous experience and suitability for the task.

3.2.1 C++

C++ has been chosen to be the main programming language for the project. This language offers a strong object oriented approach which is well suited for incremental development over an extended period of time. It also offers fast execution times compared to other languages which will be valuable when producing a simulation that should run in real time.

3.2.2 Development Environment

An integrated development environment (IDE) will be used to aid development of the project. Visual Studio 2022 offers C++ support and a range of tools to boost productivity when creating software applications. Visual Studio also contains features that

monitor resource usage at runtime, which will make conducting experiments into the application's processor and memory efficiency more straightforward. One downside of Visual Studio is it does not contain official support for developing in GLSL.

3.2.3 OpenGL

OpenGL is an API to graphics hardware. The API allows a programmer to specify the shader programs, objects and operations involved in producing high-quality graphical images, specifically colour images of three-dimensional objects (Segal and Akeley, 2022). OpenGL has been chosen over other APIs (DirectX, Vulkan) due to familiarity and execution speed, as well as it being frequently used for developing programs with similar contexts, such as those identified in Section 2.

3.2.4 GLSL, GLM and FreeGLUT

The OpenGL Shading Language (GLSL) will be the exclusive shading language for the project. It is used to specify how objects and their characteristics (i.e, shape, colour, texture etc.) are displayed. Previous experience using it to develop 3D graphical applications and extensive documentation mean it is a good choice for the project.

The OpenGL Mathematics (GLM) library, designed specifically for GLSL, offers a variety of classes and functions that are useful in developing graphical applications such as matrix transformations and vector mathematics.

Finally, FreeGLUT is an open-source OpenGL toolkit library which provides the user ways to create and manage windows in which OpenGL contexts are running. It is an updated and supported replacement for the now obsolete GLUT (OpenGL Utility Toolkit).

3.3 Set Backs and Re-Specification

Due to personal circumstances, the development rate at this stage of the project has slowed dramatically. Because of this, it felt appropriate to rethink the expectations of the project and as such go back to the original specification to see what can be met in the time constraints.

Given the complexity of Computational Fluid Dynamics, and it's underlying Navier-Stokes equations, developing a system based on fluid characteristics like viscosity, den-

sity and direction of current velocity vector field is likely to not be completed during the time frame, and so the project would not fulfill its main goal of creating a realistic visualisation of a waterfall. In addition, other elements of the simulation such as terrain generation, which is relatively simple to implement, will receive more development time if the particle model is simpler.

As such, the fluid model will now focus on producing more simple particles that follow a realistic flow path, rather than modelling their movement based on fluid and world variables. A variation element will give the path the particles take a modifiable degree of randomness in order to improve realism. This approach should allow for the potential of more realistic fluid behaviours to be implemented if there is time, whilst still meeting some specifications.

3.4 Development Plan

With this in mind, the following shows a proposed lo-fi development path for the system. These stages should allow for a key element of the system to fully implemented before work on the next stage is started.

Stage 1 - Application Setup

- Set up application for development.
- Import OpenGL, GLM, FreeGLUT and any other libraries or modules needed, and configure.
- Define methods for OpenGL render loop.
- Setup window using FreeGLUT
- Create basic vertex and fragment shader for debugging.

Stage 2 - Particle Class

- Define particle class to generate particle objects.
- Include position, size, level(defines particle roundness) and other necessary attributes.
- Define methods for creating geometry, rendering etc.

Stage 3 - Particle System Class

- Define particle system class to manage a collection of particles.
- Include particle storage container, number of particles, current particle count and other necessary attributes.
- Define methods for populating the system, rendering and removing particles etc.

Stage 4 - Particle Movement and Variation

- Define methods in the particle and particle system class that generate random flow paths.
- Define methods for randomly generating values given a supplied variation value.
- Apply randomness to flow paths to emulate waterfall behaviour.

Stage 5 - Update Lighting

- Define constants for light position, colour and strength.
- Replace basic shaders with more realistic Phong Illumination model.

Stage 6 - Terrain Generation

- Create or source 3D models to use.
- Using a 3D modelling tool, build environment using sourced models.
- Export terrain and implement methods to load to application.

Stage 7 - Collision Detection and Response

- Implement collision detection between particles and environment.
- Develop collision response based off impact angle and particle speed.
- Implement collision response.

Improvements and Refinement

The final stage during the development process would be to implement additional features and polish existing ones if the time constraints allow. Features of priority include:

- Surface generation to create a bounding polygon around the particles to represent the surface of the water.
- Redesigning the particle class to incorporate Navier-Stokes equations for modelling more realistic behaviours.
- Implementing shaders that more accurately represent how light interacts with different materials present in the scene.

3.5 UML Diagram

The following shows a preliminary UML diagram for the class structure of the application. It should be noted that attributes, methods and classes may be added, modified or removed during development, and so this diagram does not represent final system.

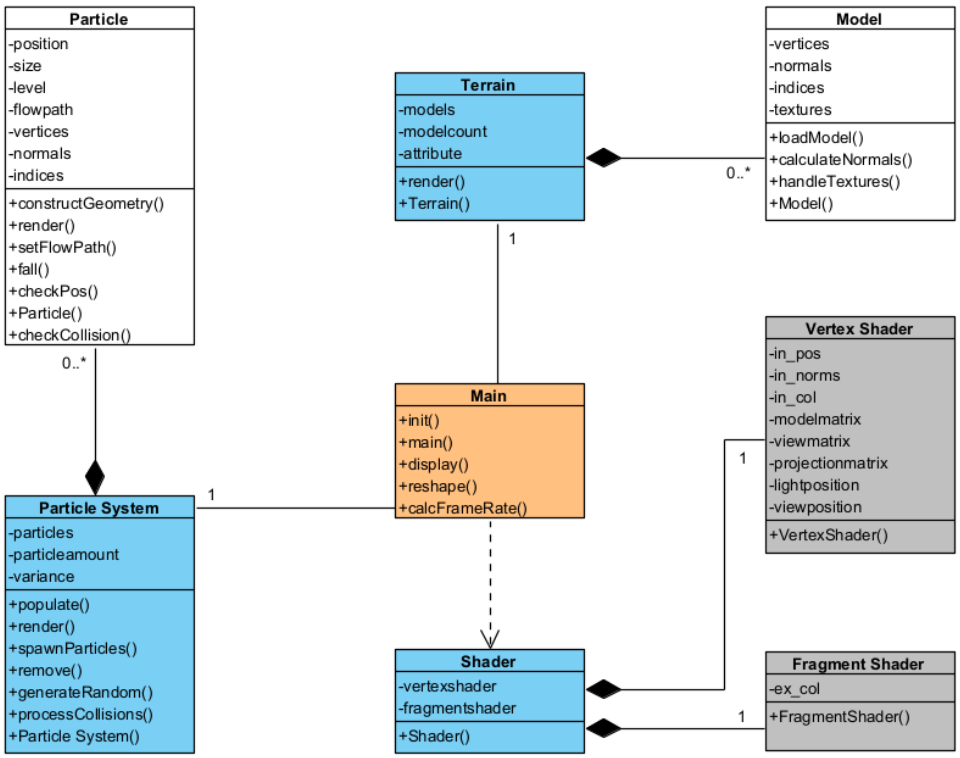

Figure 4: Preliminary UML diagram.

3.6 Particle Movement

In the real world, no two naturally occurring waterfalls are the same. Environmental differences cause each waterfall's structure, flow path and flow rate to be completely unique. This section of the report describes the research and design into producing realistic particle movement for the waterfall simulation.

3.6.1 Flow Paths

Plum defines the shape of waterfalls into 8 different topologies, each characterised by the direction, shape and amount of water that flows through it (Plumb, 2013). Examples of the 8 different topologies are shown below.

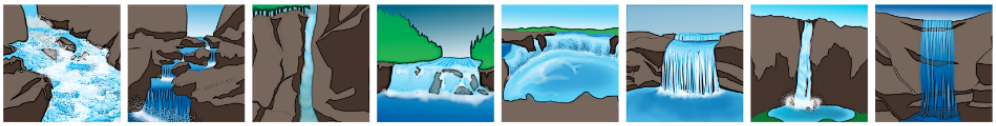

Figure 5: Waterfall topologies (Emilien et al., 2015).

In order for the waterfall simulation to appear realistic, particles must conform to a similar path that real water take during movement through a waterfall. Due to the fact that the project's time limitations doesn't allow for the implementation of true dynamic water behaviour, the easiest way to model the flow paths of the particles is to represent them via a graphing function.

Given the aim to create a waterfall that looks as realistic as possible, a waterfall topology that's flow path can be most accurately represented through a graphing function should be chosen. Because of their inherent randomness due to water flowing around obstacles and descending irregularly, topologies such as rapids and cascades should be avoided.

Similarly, cataract topolgies contain multiple different waterfalls that each descend in irregular fashions. Whilst this could be implemented, creating multiple waterfalls each with differing flow paths is likely to take development time away from other key areas of the project. This time could be better spent developing the realism of a waterfall with a simpler flow path.

For this reason; topolgies such as horsetail, block, ledge, plunge and ribbon all represent potential waterfall topologies with equally graphable flow paths. Because of this,

the fact that the topology lends itself well to potential modifications, and that it is personally most visually appealing, the plunge topology was chosen to be modelled.

Examples of plunge waterfalls were then researched. Below shows the stages in creating a graphing function that best suited the flow path of waterfall in sample images.

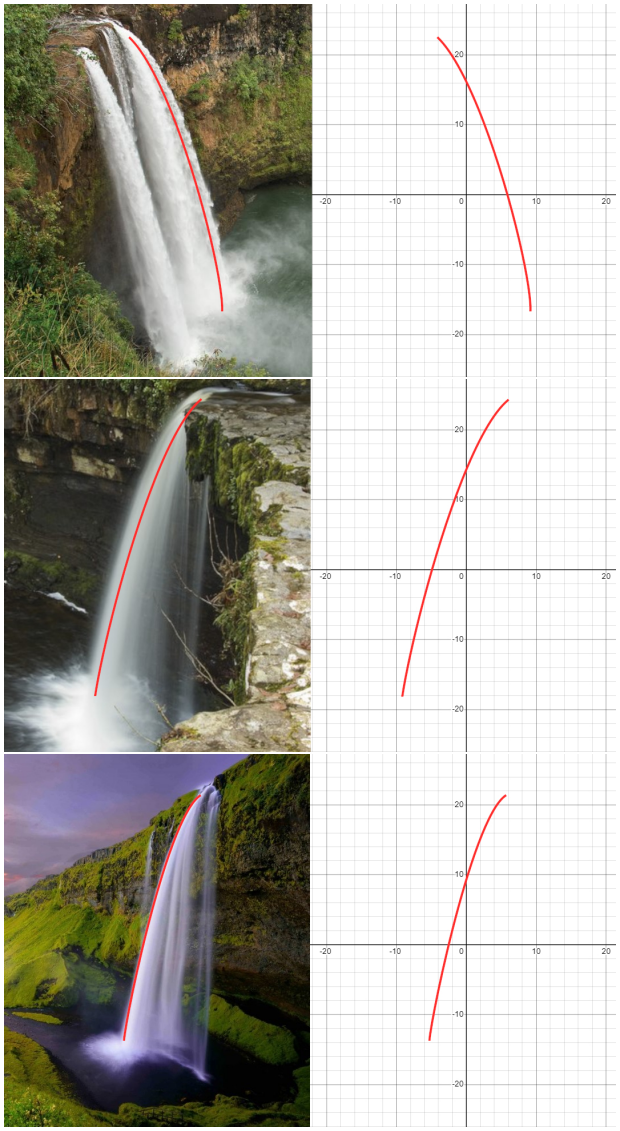

Figure 6: Graphing of waterfall flow paths.

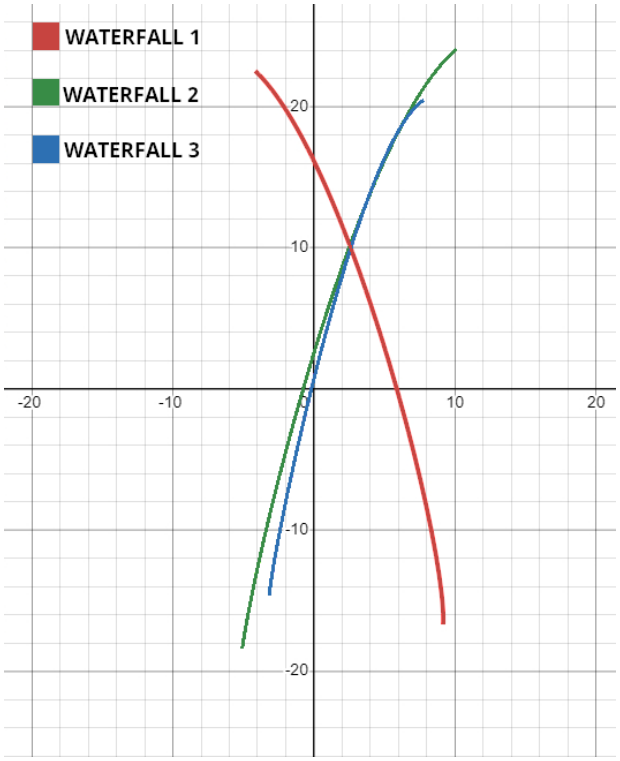

Figure 7: Graph of combined flow paths.

The flow path obtained from each waterfall image was plotted using a graphing calculator. As expected, all three paths show sections of a parabolic curve representative of a mass with horizontal velocity acting under the vertical force gravity.

Assuming air resistance is negligible, a particle of water falling from the top of a travels at a constant velocity in the horizontal direction, and so has no acceleration (Moebs et al., 2016). Therefore the particle's horizontal position in the flow path is given as:

$$v_{0x} = v_x, \quad x = x_0 + v_x t \tag{2}$$

Where v_x is the horizontal velocity, x is the horizontal position and t is time surpassed since start of fall. Again assuming negligible air resistance, the equation for the particle's vertical position in the flow path is given as:

$$v_{0y} = v_{0y} - gt, \quad y = y_0 + v_{0y}t - \frac{1}{2}gt^2 \tag{3}$$

Where v_y is the vertical velocity, y is the vertical position, g is acceleration due to gravity ($\approx 9.81ms^{-2}$) and t is time surpassed since start of fall. We can then rearrange 2 to become:

$$t = \frac{x - x_0}{v_x} \tag{4}$$

The starting point of the particle can be given as zero relative to it's current position, therefore $x_0 = 0$ and $y_0 = 0$, and so can be discounted from 3 and 4. Substituting 4 into 3 becomes:

$$y = v_{0y}\frac{x}{v_x} - \frac{1}{2}g(\frac{x}{v_x})^2 \tag{5}$$

Finally, v_{0y} and v_x can be seen as constants (and so can be discounted from 5) given that:

1. There is no change in horizontal velocity during the fall.
2. No other vertical forces were acting on the particle when $t = 0$.

And so for any given position of x the corresponding y value on the flow path is given as:

$$y = x - \frac{1}{2}gx^2, \quad \text{where } x > 0. \tag{6}$$

This equation allows us to model the path of a falling water particle in the x and y axis. However, this does not provide a position in three dimensions which is required in the specification of the project. Nevertheless, because the z axis of an object also represents movement along the horizontal axis, velocity in this direction is constant and so can be represented by a value that also changes at a constant rate; namely x.

3.6.2 Variation

Giving each particle an element of randomness to its flow path should aid in increasing how realistic the waterfall simulation is visually. To do this, variables can be added to the flow path equation that transform the graph in various ways. Demonstrating this, a, b and c are added to the equation. To show this more clearly, 6 can be factorised and re-written as:

$$y = a(x+b)^2 + c, \quad \text{where } a \in (\mathbf{Q} < 0) \text{ and } b, c \in \mathbf{Q} \tag{7}$$

Following the constraints set out in 7, assigning random values to a, b and c for will ensure that each particle's flow path is slightly different whilst maintaining a sense of realism. Graphs illustrating this are shown below. Arbitrary values have been selected for a, b and c in both cases to show how they influence the flow path.

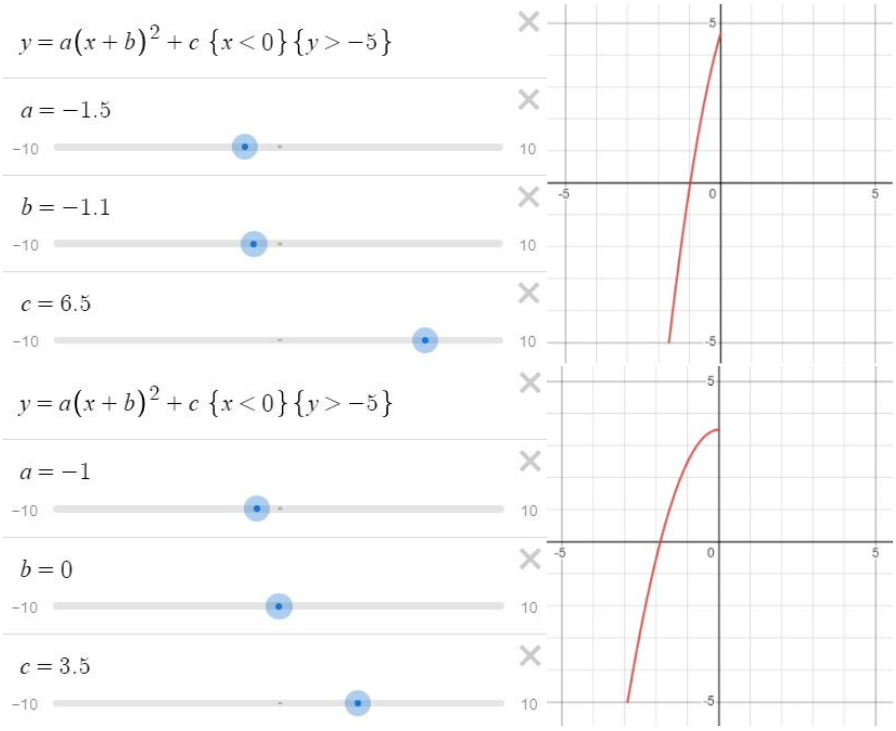

Figure 8: Graphing of waterfall flow paths.

3.7 Lighting Model

Another essential aspect of the system is the lighting model that will be used to shade objects in the scene, which will mainly consist of water particles and terrain. In order to do this, the Phong Illumination Model will be used to colour object's surfaces dependent on the colour, strength and angle of the light incident to it (Phong, 1975). The Phong Illumination Model combines three types of light sources in order to calculate the sum of the intensity of light incident on the surface of an object. The three components

are described below, with explanations sourced from Laycock's lectures on the subject (Laycock, 2021).

Ambient

The ambient component of the model represents the sum of all light sources and all light reflected from all surfaces in a scene. It provides a low intensity, non-directional "ambient" amount of light to the objects in a scene. The ambient component simply provides a uniform illumination of an object and so alone does not produce a very realistic lighting model.

Diffuse

The diffuse element adds a component on top of the ambient element. This element models the interaction of a light ray when incident upon a dull, matte surface. The reflection of the incident light on the surface is known as Lambertian reflection.

The brightness that appears on a Lambertian surface is dependent on the direction of the incident light ray and the surface normal, as shown in the diagram below. The vector l represents the direction of the incident light ray and n shows the normal to the surface.

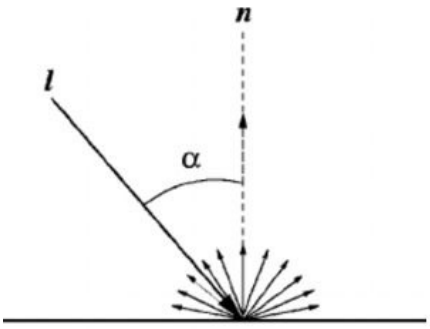

Figure 9: Light incident on a Lambertian surface (Aytac and Barshan, 2005).

Specular

Reflective surfaces exhibit points of almost maximum brightness, and these points are known as specular reflections. These reflections are the final component of the

Phong Illumination model and combined with the previous two elements in order to produce a realistic lighting model.

The position of the specular component is dependent on the angle between the light ray incident to the surface and the normal to the surface at the point of incidence. The strength of the component, i.e. how bright it appears, is instead dependent on the angle between the reflected light ray and the viewing direction. The diagram below shows the components that make the specular element of the lighting model.

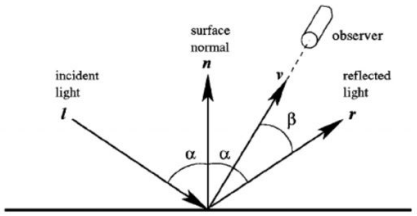

Figure 10: Light incident on a reflective surface (Aytac and Barshan, 2005).

The vectors representing v and r are compared by taking the cosine of their dot product. The more similar the directions of v and r are, the lower the angle between them and so the intensity of the specular highlight is greater.

Lastly, the strength of reflected light follows an inverse square law to distance, and so an attenuation coefficient is added to the model to simulate the intensity of light incident on a surface decreasing the further the object is from the light source.

The diagram below show the individual elements of the Phong Illumination model and how they combine to produce a realistically shaded object.

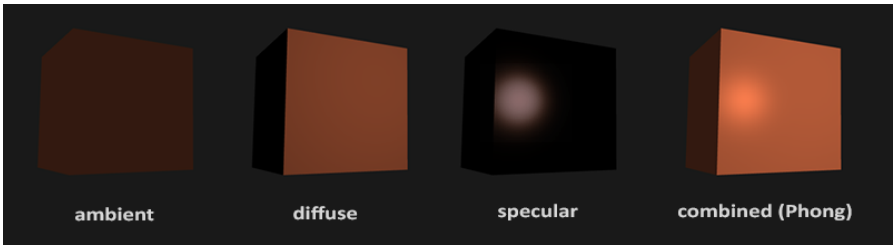

Figure 11: Phong Illumination Model (LearnOpenGL, 2014)

4 Implementation

Once the system was designed, implementation was started. The following section shows the code used to implement areas of notable importance to the functionality of the simulation.

4.1 Initialisation

```
void init()
{
glClearColor(0.07f, 0.13f, 0.17f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT);

if(!myShader.CreateShaderProgram("Basic␣Shader␣w/␣Phong",
"glslfiles/basicSpecular.vert", "glslfiles/basicSpecular.frag"))
{
cout << "failed␣to␣load␣shader" << endl;
}

glUseProgram(myShader.GetProgramObjID());

glUniform4fv(glGetUniformLocation(myShader.GetProgramObjID(),
"LightPos"), 1, LightPos);
glUniform4fv(glGetUniformLocation(myShader.GetProgramObjID(),
"light_ambient"), 1, Light_Ambient);
glUniform4fv(glGetUniformLocation(myShader.GetProgramObjID(),
"light_diffuse"), 1, Light_Diffuse);
glUniform4fv(glGetUniformLocation(myShader.GetProgramObjID(),
"light_specular"), 1, Light_Specular);

glUniform4fv(glGetUniformLocation(myShader.GetProgramObjID(),
"material_ambient"), 1, Material_Ambient);
glUniform4fv(glGetUniformLocation(myShader.GetProgramObjID(),
"material_diffuse"), 1, Material_Diffuse);
glUniform4fv(glGetUniformLocation(myShader.GetProgramObjID(),
"material_specular"), 1, Material_Specular);
glUniform1f(glGetUniformLocation(myShader.GetProgramObjID(),
"material_shininess"), Material_Shininess);

waterfall.populate();
```

```
glEnable(GL_DEPTH_TEST);
}
```

4.2 Main Render Loop

```
void display()
{

calculateFrameRate();

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glUseProgram(myShader.GetProgramObjID());

// Send projection, view, normal and modelview matrices to Shader
glUniformMatrix4fv(glGetUniformLocation(myShader.GetProgramObjID(),
"ProjectionMatrix"), 1, GL_FALSE, glm::value_ptr(ProjectionMatrix));

//SideView
//ViewMatrix = glm::translate(glm::mat4(1.0), glm::vec3(0, 25, -65));

//FrontView
ViewMatrix = glm::rotate(glm::mat4(1.0), glm::radians(-90.0f),
glm::vec3(0.0, 1.0, 0.0));
ViewMatrix = glm::translate(ViewMatrix, glm::vec3(-65, 25, 0));

glUniformMatrix4fv(glGetUniformLocation(myShader.GetProgramObjID(),
"ViewMatrix"), 1, GL_FALSE, glm::value_ptr(ViewMatrix));
NormalMatrix = glm::inverseTranspose(glm::mat3(ModelViewMatrix));
glUniformMatrix3fv(glGetUniformLocation(myShader.GetProgramObjID(),
"NormalMatrix"), 1, GL_FALSE, glm::value_ptr(NormalMatrix));
ModelViewMatrix = glm::translate(ViewMatrix,glm::vec3(0,0,0));
glUniformMatrix4fv(glGetUniformLocation(myShader.GetProgramObjID(),
"ModelViewMatrix"), 1, GL_FALSE, glm::value_ptr(ModelViewMatrix));

// Object rendering
waterfall.render();

glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

```
glUseProgram(0);

glutSwapBuffers();
glutPostRedisplay();
}
```

4.3 Generating Sphere Geometry

```
//create the memory for the Particle
verts = new float[((level-2)* level+2)*3];
norms = new float[((level-2)* level+2)*3];
tInds = new unsigned int[(((level-3)*(level-1) + (level-1)) * 2)*3];

numOfTris = (((level-3)*(level-1) + (level-1)) * 2);
numOfVerts = ((level-2)* level+2);

//populate the arrays
float theta, phi;
int vertCount = 0;
int i, j, t;

for( t=0, j=1; j<level-1; j++ )
{
    for(       i=0; i<level; i++ )
    {
        theta = float(j)/(level-1) * PI;
        phi   = float(i)/(level-1 ) * PI*2;

        verts[vertCount+t] = ((sinf(theta) * cosf(phi))*r)+cx;
        verts[vertCount+t+1] = (cosf(theta)*r)+cy;
        verts[vertCount+t+2] = ((-sinf(theta) * sinf(phi))*r)+cz;

        norms[vertCount+t] = (sinf(theta) * cosf(phi));
        norms[vertCount+t+1] = cosf(theta);
        norms[vertCount+t+2] = -(sinf(theta) * sinf(phi));

        t+=3;
    }
}
verts[vertCount+t] = cx;
```

```
verts[vertCount+t+1] = r+cy;
verts[vertCount+t+2] = cz;

norms[vertCount+t] = 0;
norms[vertCount+t+1] = 1;
norms[vertCount+t+2] = 0;
t+=3;

verts[vertCount+t] = cx;
verts[vertCount+t+1] = -r+cy;
verts[vertCount+t+2] = cz;

norms[vertCount+t] = 0;
norms[vertCount+t+1] = -1;
norms[vertCount+t+2] = 0;
t+=3;

int vc3 = vertCount / 3;
int triCount = 0;
for( t=0, j=0; j<level-3; j++ )
{
    for(       i=0; i<level-1; i++ )
    {
        tInds[triCount+t] = vc3 + ((j  )*level + i);     t++;
        tInds[triCount+t] = vc3 + ((j+1)*level + i+1);     t++;
        tInds[triCount+t] = vc3 + ((j  )*level + i+1);     t++;
        tInds[triCount+t] = vc3 + ((j  )*level + i ) ;     t++;
        tInds[triCount+t] = vc3 + ((j+1)*level + i ) ;     t++;
        tInds[triCount+t] = vc3 + ((j+1)*level + i+1);  t++;
    }
}
for( i=0; i<level-1; i++ )
{
    tInds[triCount+t] = vc3 + ((level-2)*level);  t++;
    tInds[triCount+t] = vc3 + (i);     t++;
    tInds[triCount+t] = vc3 + (i+1);     t++;
    tInds[triCount+t] = vc3 + ((level-2)*level+1);     t++;
    tInds[triCount+t] = vc3 + ((level-3)*level + i+1);     t++;
    tInds[triCount+t] = vc3 + ((level-3)*level + i);  t++;
}
triCount += t;
```

4.4 Vertex Shader

```
// Vertex Shader

#version 430

uniform mat4 ModelViewMatrix;
uniform mat4 ProjectionMatrix;
uniform mat4 ViewMatrix;
uniform mat3 NormalMatrix;
//inverse transpose of the 3x3 bit of modelview matrix
uniform vec4 LightPos;  // light position

in  vec3 in_Position;  // Position coming in
in  vec3 in_Normal;    // vertex normal used for lighting
out vec3 ex_Normal;    // exiting normal transformed by the normal matrix

out vec3 ex_LightPos;
//light direction vector
in eye space (assuming it doesn't undergo other transformations)
out vec3 ex_PositionEye;
//vertex position in eye space (i.e. after ModelView but not projection)

void main(void)
{
gl_Position = ProjectionMatrix * ModelViewMatrix * vec4(in_Position, 1.0);

ex_Normal = NormalMatrix*in_Normal;

ex_PositionEye = vec3((ModelViewMatrix * vec4(in_Position, 1.0)));

ex_LightPos = vec3(ViewMatrix*LightPos);
}
```

4.5 Fragment Shader

```
// Fragment Shader

#version 430
```

```
out vec4 out_Color;
//colour for the pixel

uniform vec4 light_ambient;
//ambient light
uniform vec4 light_diffuse;
//diffuse property for the light
uniform vec4 light_specular;
//specular property for the light

uniform vec4 material_ambient;
//ambient property for material
uniform vec4 material_diffuse;
//diffuse property for the material
uniform vec4 material_specular;
//specular property for the material
uniform float material_shininess;
//shininess exponent for specular highlight

in vec3 ex_LightPos;
//light direction arriving from the vertex
in vec3 ex_Normal;
//normal arriving from the vertex
in vec3 ex_PositionEye;

void main(void)
{
vec4 color = light_ambient * material_ambient;

vec3 n, L;
float NdotL;

n = normalize(ex_Normal);
L = normalize(ex_LightPos - ex_PositionEye);

vec3 v = normalize(-ex_PositionEye);
vec3 r = normalize(-reflect(L, n));

float RdotV = max(0.0, dot(r, v));

NdotL = max(dot(n, L),0.0);
```

```
if(NdotL > 0.0)
{
    color += (light_diffuse * material_diffuse * NdotL);
    color += light_specular * material_specular * pow(RdotV, material_shininess);
}

out_Color = color;
}
```

4.6 Random Flow Path Constant Generation

```
float ParticleSystem::generateRand(float variance)
{

// Generate a float between -var and +var
uniform_real_distribution<float> dis(-variance, variance);

return dis(eng);
}

// Returns a random float between -variance * spread and +variance * spread.
float ParticleSystem::generateRandZ(float variance)
{
float spread = 1.0f;
uniform_real_distribution<float> dis(-variance*spread, variance*spread);

return dis(eng);
}

// Returns a random float between 1.5 and +variance.
float ParticleSystem::genRandCoefficient(float variance)
{

uniform_real_distribution<float> dis(1.5, variance);

return dis(eng);
}
```

4.7 Flow Path Calculation

```
float Particle::calcY(float x)
{
float y = a * (-pow((x + b), 2) + c);
return y;
}

void Particle::fall()
{
cx -= 0.1;
cy = calcY(cx);
cz += z * 0.1;
}
```

5 Experimental Results

The following section details the experiments that were carried out to determine the performance of the simulation. The tools used to collect the data during these experiment on resource useage were from Visual Studio's integrated tool set.

5.1 Hardware

The performance of graphical applications are highly dependent on the hardware that is present in the physical system that they run on. Because of this, it is important to note the hardware these experiments were run on in order to give a frame of reference for comparisons to other implementations of simulating a waterfall.

CPU: Intel Core i7-6700k @ 4.0Ghz

GPU: Nvidia Geforce GTX 980Ti

Memory: 16.0GB DDR4 @ 2400 MHz

Operating System: x64 Windows 10

5.2 Particle Count

During this experiment, the amount of particles concurrent in the scene was incremented in regular intervals and the effect on average frames per second displayed, memory useage and processor useage was measured. The particle quality, i.e. how many surfaces a particle is formed by, was kept constant (5).

The visual impact of the change in particle count is also shown.

Table 1: Results from investigations into the impact of increasing the number of particles on system performance.

Particle count (n)	Average frames per second (n)	Peak memory useage (mb)	Peak processor useage(%)
0	605	41	14
100	370	587	15
200	183	1100	20
300	126	1500	23
400	93	1800	23
500	76	2400	23
600	64	2800	25
700	54	3100	25
800	46	3000	23
900	42	3300	25
1000	38	3300	25
1100	34	3600	25
1200	32	3400	25
1300	29	3300	25
1400	27	3300	25
1500	25	3600	25

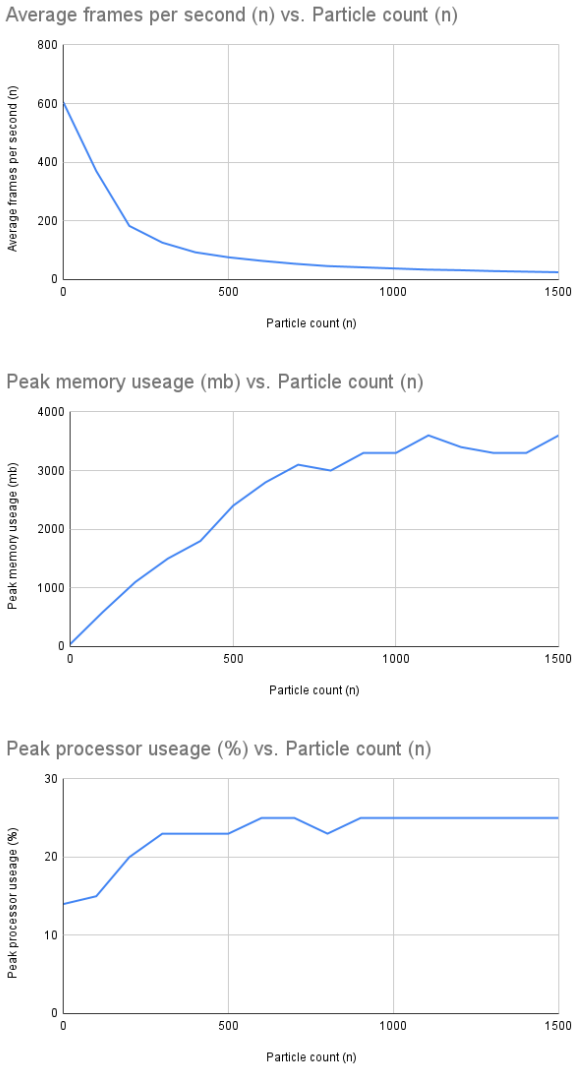

Figure 12: Graphing of results of investigating increasing particle count on performance.

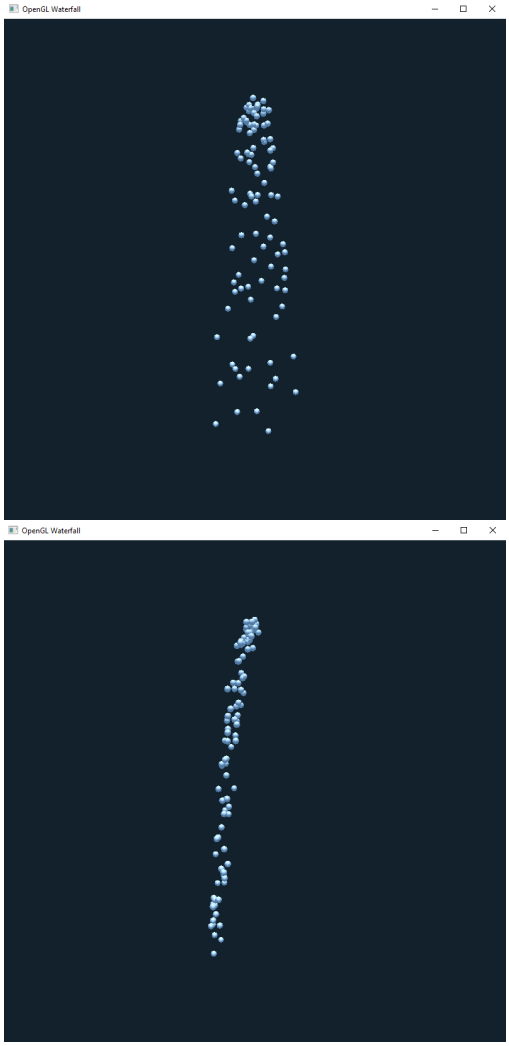

Figure 13: Low particle count (front and side view).

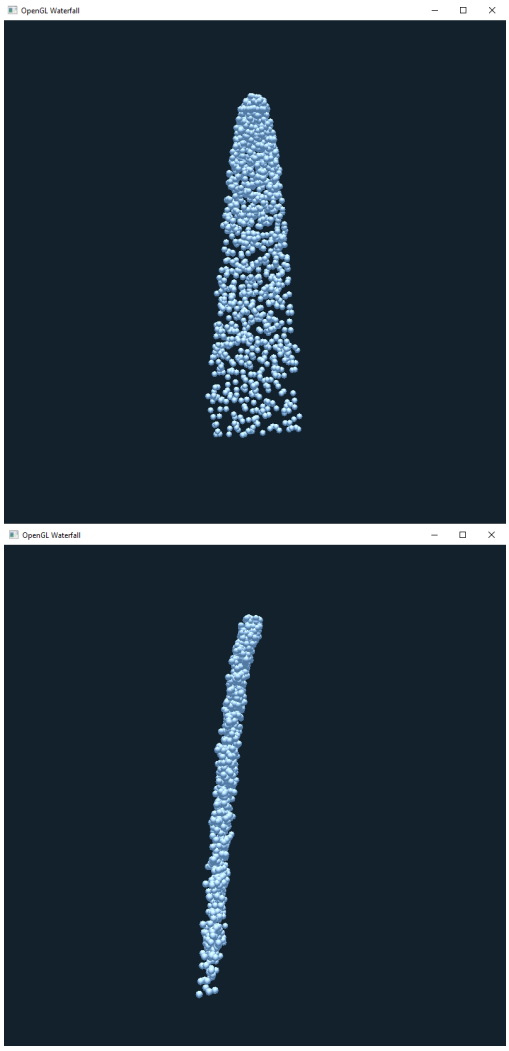

Figure 14: High particle count (front and side view).

5.3 Particle Quality

In this experiment, the quality of each rendered particle (i.e. how many surfaces it is formed by) is incremented in regular intervals and the effect on average frames per second displayed, memory useage and processor useage was measured. The number of particles concurrent in the scene was kept constant (250).

The visual impact of the change in particle quality is also shown.

Table 2: Results from investigations into the impact of increasing the quality of each particle on system performance.

Particle quality (n)	Average frames per second (n)	Peak memory useage (mb)	Peak processor useage(%)
5	150	1200	19
10	136	1400	20
15	114	1400	22
20	96	1500	24
25	77	1800	25
30	58	2100	26
35	43	2400	27
40	34	2800	28
45	27	3400	29
50	22	3700	29

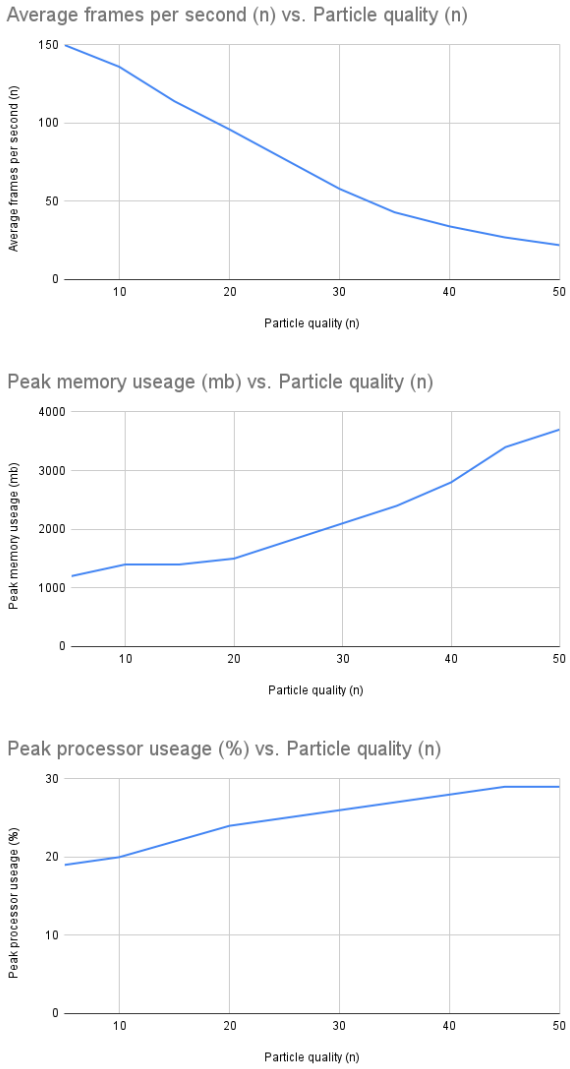

Figure 15: Graphing of results of investigating increasing particle quality on performance.

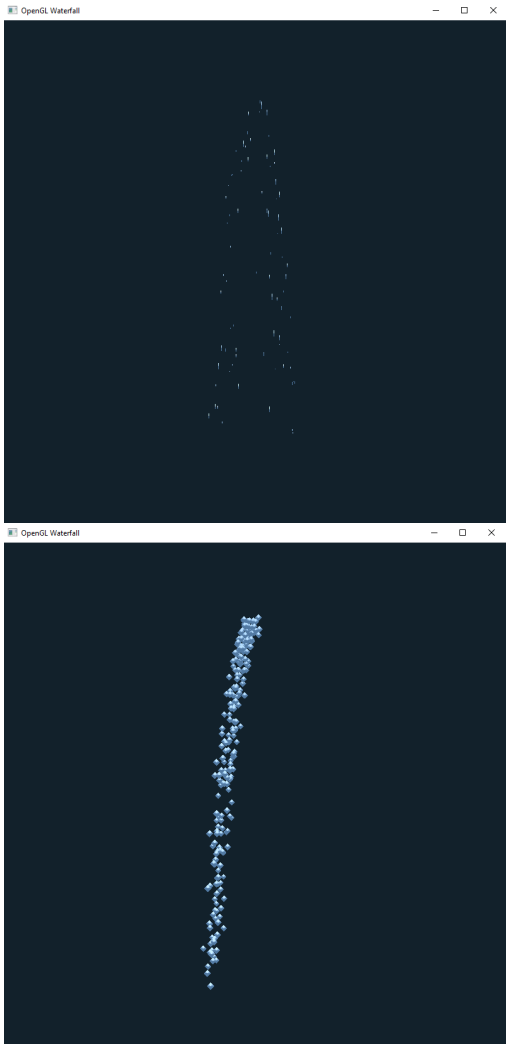

Figure 16: Low particle quality (front and side view).

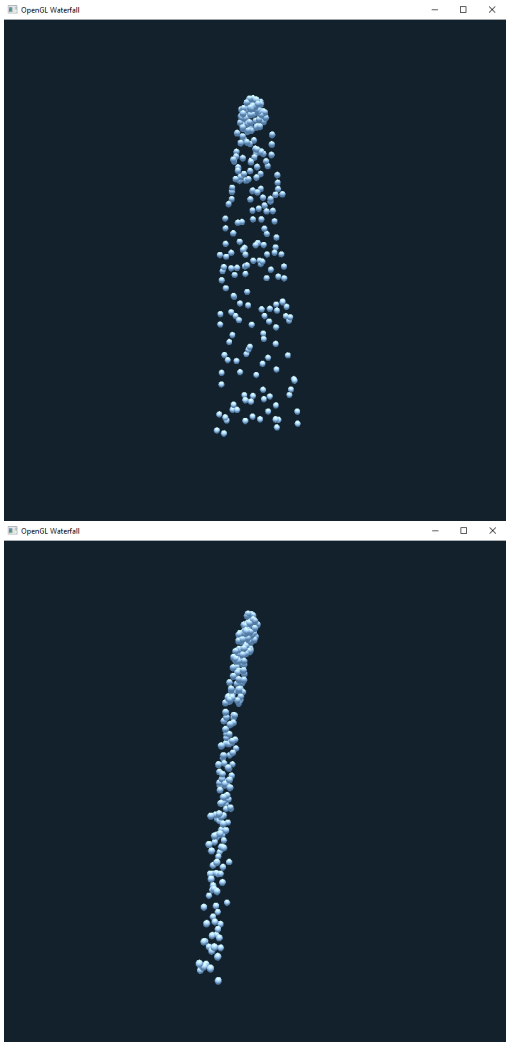

Figure 17: High particle quality (front and side view).

6 Discussion and Evaluation

The experiments conducted on the produced results that were generally expected. As shown in Table 1 and subsequent graphs in Figure 12, increasing the amount of particles concurrently rendered to the screen decreases the average frame rate. This is likely due to the increased time required to complete all positional calculations for the particle system before the next frame can be rendered.

Similarly, as the particle count increased, the larger the amounts of memory and pro-

cessing resources have to be allocated to the application in order to render the frame to the screen.

Interestingly, the frame rate did not decrease at a rate that was linear to the decrease in particle count. Graph 1 in Figure 12 shows that increments in particle count beyond n=250 result in smaller decreases in the frame rate the system ran at. Another point worth noting is the unexpected plateau in peak processor useage once n>1000, whereas before this a generally liner pattern is shown.

In general, the accuracy of the visual representation of the waterfall increases with particle count up to a certain point, with this level being identified at around n=400. Beyond this increasing the particle count shows particular increase in realism.

This is a similar case with particle quality. Figure 16 shows that at low values of particle quality the waterfall appears less realistic. This behaviour is most noticeable when viewed from the front of the waterfall. However, particle quality over n=5 doesn't appear to provide any greater sense of realism to the scene, potentially due to the small size of the individual particles which makes it harder to see changes in quality.

Finally, increasing the particle quality was overall more consistent in decreasing he systems performance than particle count was, as shown in Figure 15, where Graph 1 shows a more linear pattern than Graph 1 in Figure 12. Interestingly enough the same plateau in peak processor useage appears for increasing particle quality too, most notably at n>45.

Whilst the system did not meet certain areas of the specification, such as generating a suitable environment for the waterfall to occupy, the waterfall's particles movement is fairly realistic to how it appears in the real world.

Nonetheless, many areas of the simulation could be improved upon. The next stages in developing the application would be to implement a truly dynamic Smoothed Particle system based off the Navier Stokes equations. In addition, surface smoothing should be prioritised when developing the system as this would allow the water in the system to appear more like one continuous fluid structure.

7 Conclusion

In summary, whilst not all the specification points have been met, the application still provides a framework on which they can be addressed in future. The elements of the waterfall that are present show realistic behaviour that is fairly representative of those

found in nature. Finally, the project has provided multiple points of learning and as such has developed my understanding behind the production of academic papers; as well as the challenges and opportunities the field of computer graphics presents.

References

Aytac, T. and Barshan, B. (2005). Surface differentiation by parametric modeling of infrared intensity scans. *Optical Engineering*, 44.

Bagar, F., Scherzer, D., and Wimmer, M. (2010). A layered particle-based fluid model for real-time rendering of water. In *Computer Graphics Forum*, volume 29, pages 1383–1389. Wiley Online Library.

Bender, J., Erleben, K., and Galin, E. (2011). Sph based shallow water simulation. In *Workshop on Virtual Reality Interaction and Physical Simulation*.

Benz, W. (1988). Applications of smooth particle hydrodynamics (sph) to astrophysical problems. *Computer Physics Communications*, 48(1):97–105.

Bluestein, D. (2017). Utilizing computational fluid dynamics in cardiovascular engineering and medicine—what you need to know. its translation to the clinic/bedside. *Artificial Organs*, 41(2):117.

Dobek, S. (2012). Fluid dynamics and the navier-stokes equation. *University of Maryland*.

Emilien, A., Poulin, P., Cani, M.-P., and Vimont, U. (2015). Interactive procedural modelling of coherent waterfall scenes. In *Computer Graphics Forum*, volume 34, pages 22–35. Wiley Online Library.

Gingold, R. A. and Monaghan, J. J. (1977). Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly notices of the royal astronomical society*, 181(3):375–389.

Gonzalez-Ochoa, C., Holder, D., and Cook, E. (2012). From a calm puddle to a stormy ocean: Rendering water in uncharted. In *ACM SIGGRAPH 2012 Talks*, page 1. Association for Computing Machinery.

Guan, Y., Chen, W., Zou, L., Zhang, L., and Peng, Q. (2006). Modeling and rendering of realistic waterfall scenes with dynamic texture sprites. *Computer Animation and Virtual Worlds*, 17:573–583.

Husni, E. M., Hamdi, K., and Mardiono, T. (2009). Particle system implementation using smoothed particle hydrodynamics (sph) for lava flow simulation. In *2009 International Conference on Electrical Engineering and Informatics*, volume 1, pages 216–221. IEEE.

Iglesias, A. (2004). Computer graphics for water modeling and rendering: a survey. *Future generation computer systems*, 20(8):1355–1374.

Johanson, C. and Lejdfors, C. (2004). Real-time water rendering. *Lund University*.

Laycock, S. (2021). Lighting and the phong illumination model. *Lecture at the University of East Anglia, 5th September*.

Lee, H. and Han, S. (2010). Solving the shallow water equations using 2d sph particles for interactive applications. *The Visual Computer*, 26(6):865–872.

Lind, S. J., Rogers, B. D., and Stansby, P. K. (2020). Review of smoothed particle hydrodynamics: towards converged lagrangian flow modelling. *Proceedings of the Royal Society A*, 476(2241):20190801.

Losasso, F., Talton, J., Kwatra, N., and Fedkiw, R. (2008). Two-way coupled sph and particle level set fluid simulation. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):797–804.

Łukaszewicz, G. and Kalita, P. (2016). Navier–stokes equations. *Advances in Mechanics and Mathematics*, 34.

Moebs, W., Ling, S., and Sanny, J. (2016). *University Physics Volume 1*. OpenStax, Houston, Texas, 1 edition.

Monaghan, J. J. (1992). Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics*, 30:543–574.

Monaghan, J. J. (2005). Smoothed particle hydrodynamics. *Reports on progress in physics*, 68(8):1703.

Monaghan, J. J. (2012). Smoothed particle hydrodynamics and its diverse applications. *Annual Review of Fluid Mechanics*, 44:323–346.

Müller, M., Charypar, D., and Gross, M. H. (2003). Particle-based fluid simulation for interactive applications. In *Symposium on Computer animation*, volume 2.

Phong, B. T. (1975). Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317.

Plumb, G. (2013). *Waterfall lover's guide pacific northwest*. Mountaineers Books, Seattle, WA, 5 edition.

Sakaguchi, R., Dufor, T., Zalzala, J., Lambert, P., and Kapler, A. (2007). End of the world waterfall setup for pirates of the caribbean 3". In *SIGGRAPH Sketches*, page 90.

Schödl, A., Szeliski, R., Salesin, D. H., and Essa, I. (2000). Video textures. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 489–498.

Segal, M. and Akeley, K. (2022). The opengl graphics system: a specification. *OpenGL Documentation Archive*, 46.

Sumner, J., Watters, C. S., and Masson, C. (2010). Cfd in wind energy: the virtual, multiscale wind tunnel. *Energies*, 3(5):989–1013.

Suwardi, M., Tarwidi, D., et al. (2018). Smoothed particle hydrodynamics method for simulating waterfall flow. In *Journal of Physics: Conference Series*, volume 971, page 012035. IOP Publishing.

Takahashi, T., Fujii, H., Kunimatsu, A., Hiwada, K., Saito, T., Tanaka, K., and Ueki, H. (2003). Realistic animation of fluid with splash and foam. In *Computer Graphics Forum*, volume 22, pages 391–400. Wiley Online Library.

van der Laan, W. J., Green, S., and Sainz, M. (2009). Screen space fluid rendering with curvature flow. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 91–98.

Wong, A. H.-K. (1994). Waterfall-a particle system animation. *Computer Science Department Cornell University*, 199:l.

Xiao, X., Zhang, S., and Yang, X. (2018). Fast, high-quality rendering of liquids generated using large-scale sph simulation. *Journal of Computer Graphics Techniques Vol*, 7(1).