Real Time Fluid Simulation & Rendering: Ocean Model

Leon Johnson
963653

Submitted to Swansea University in fulfilment
of the requirements for the Degree of Master of Science

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

April 28, 2021

Declaration

This work has not previously been accepted in substance for any degree and is not being concurrently submitted for any degree.

Signed Leon Johnson (candidate)

Date 28/04/2021

Statement 1

This dissertation is being submitted in partial fulfillment of the requirements for the degree of MSci Computer Science.

Signed Leon Johnson (candidate)

Date 28/04/2021

Statement 2

This dissertation is the result of my own independent work / investigation, except where otherwise stated. Other sources are specifically acknowledged by clear cross referencing to author, work and page(s) using the bibliography / references section. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this dissertation and the degree examination as a whole.

Signed Leon Johnson (candidate)

Date 28/04/2021

Statement 3

I hereby give consent for my dissertation to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organizations.

Signed Leon Johnson (candidate)

Date 28/04/2021

I would like to dedicate this work to my family and the discord boys (FaZe Clan) for pushing me everyday to become the best Dota 2 player in Swansea (6k2 mmp)

Abstract

Fluid Simulation has been incorporated into modern computer graphics with application in the fields of computational fluid dynamics - to solve engineering problems - as well as to enhance perceived experience within the gaming and cinematic industries. As hardware and software has evolved, there has been an increased demand for such simulations and renders to be computed in real-time rather than pre-computed. Real-time simulation and rendering methodologies were explored in this paper, where an ocean simulation model was identified as the primary topic of investigation and to implement the project. Implementation of the ocean model was generated using Fast Fourier Transform driven methods computed in parallel on a GPU to generate a wave heightfield. Properties were extracted from the heightfield to be used as input for the render program, which was used to visualise the simulation. It was found that when the model was implemented on a NVIDIA 'Turing' GPU with compute capability of 7.5, frame rates were deliverable in excess of 144 fps on a mesh size of 1024x1024. However, when implemented on the same device with an expanded mesh size of 2048x2048, frame rates of 88 fps were attained; suggesting that computational expense scales directly with mesh size used within the model. It was also discovered that the rendering technique of Blinn-Phong shading can produce semi-realistic results on the ocean model, however there are alternative approaches such as Fresnel techniques which produce more realistic visual results. The combination of the ocean simulation technique and rendering model reserves enough computational power to be adequate for use within a larger application. This would be beneficial for an application such as a video game where the ocean model is not the primary feature, where resources can be delegated elsewhere without sacrificing realism and performance. Moving forward alternative rendering approaches can be used to utilize the surplus computational power to produce a more realistic visual output. Additional strain can be placed within the scene such as objects on the fluid model to greater test the efficiencies of the simulation when external computation is occurring.

Acknowledgements

Huge appreciation to my Mum for supporting me, tolerating me and giving me the best upbringing, something I can never repay. Appreciation to Dr. Sean Walton for his support, patience and guidance during the process of this work-piece and to Dr. Stephen Lindsay. Endless love for my gran and grampa for everything they have taught me and done for me in my life. Special thanks to Karen Baker and Pat Jagus for my previous education. Thanks to everyone and everything I have encountered in my life. Teish is dumb.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivations

The phenomenon of fluid behaviour is observed in our everyday lives. Computer graphics has been used as a tool to replicate and also gain greater insight to these behaviours by simulation and visualization, which is what this paper explores - an implementation of a fluid simulator and renderer.

Visual art exists to depict images of scenes, done so using any available medium . In the modern era, the invention of computer graphics has provided artists with tools capable of synthesising still art scenes, but also scenes consisting of multiple frames, providing the illusion of a moving image.

Computer graphics is present in a large field of modern media, whether it be art, video games, movies or marketing. However as technology has evolved, so has the demand for visual accuracy in graphical representations to provide increasingly realistic experiences. This creates an overarching challenge for graphics programmers, as with visual accuracy comes the expenditure of computational power, so the battle arises in achieving the most true and realistic image within the computational constraints. The type of rendering used for the media affects how much of a problem this is to the programmer and can be identified as two distinct sub-sections of computer graphics. *Real-Time Rendering (Online Rendering*) where the output from the render is required within milliseconds, this is "the most highly interactive area of computer graphics" [1] and also the technique deployed in video games. Or *Pre-Rendering (Offline Rendering)* where renders are photo-realistic, as visual quality is the priority rather

than performance and interactivity.

In this piece of work *Real-Time Rendering* will be of focus. In order to attain the render speeds necessary, a sacrifice has to be made in the true visual accuracy of the model, these sacrifices are made in the equations to simulate behaviours and also in the technique used to render. This spawns a multitude of problems within this field, as it is difficult to efficiently replicate certain phenomena that exists in real life to a computer program.

An example of this is the behaviours and appearances of fluids (liquids and gases), which have complex interactions with their surroundings which make it difficult to accurately simulate both visually and physically, but also exist in different forms, i.e water droplets in rain, fast flowing water in a river, waves in an ocean. The appearance of fluids in the scene can make or break the immersion for the end user, but if done correctly can produce beautiful results. Many modern video game studios are able to effectively model water in their engines to provide artists the ability to create stunning scenes, such can be seen in Naughty Dog's acclaimed title *The Last of Us Part II*. (See Figure 1.1)

Figure 1.1: Naughty Dog Exhibiting Water in a Scene in The Last of Us Part II

As well as existing as part of the backdrop, fluids can be used as the centrepiece of a game, perhaps a sailing game where interaction of the waves affects the player and needs to be accurate such as in Xbox Game Studios' *Sea Of Thieves* (See Figure 1.2) [1], or even something subtle such as a character drinking water from a glass. These different scenarios sometimes require different approaches to simulating its behaviour, creating more depth to this field.

[1]Image Source: https://www.gamespot.com/articles/sea-of-thieves-will-now-be-easier-for-players-with/1100-6480375/

Figure 1.2: Sea of Thieves Ocean Water in an Open World Environment

1.2 Objective

In this piece of work, the simulation and rendering of fluids in real time was investigated, working around the challenge of attaining visual accuracy whilst reserving as much computational power possible. The identification of a fluid modelling technique will be required in order to study and implement the related methodologies. The result of the implementation will then be analysed and compared to similar solutions to explore avenues in which the solution can be refined to increase efficiency and/or visual accuracy.

More concretely the objectives for this project are as follows:

- **Implement a program capable of efficiently simulating a fluid**

  The program will be required to run a stable simulation of a fluid, changing its state with respect to time.

- **Implement a program capable of rendering a fluid**

  The program will be required to create a visualization of the fluid simulation.

- **Analyse the implemented solution and compare with similar solutions**

  Measuring the computational expense of the implementation, and comparing alongside similar solutions both visually and computationally.

- **Identify areas of the solution that could be improved and/or optimized**

Using information from analysis and comparison to other implementations, improve on the current solution to achieve better results both visually and computationally.

Chapter 2

Related Work

The following chapter analyses and reviews the focal literature that exists in both simulating and rendering fluids. Work discussed here will be used as a foundation for this work piece and inspire the approach. Reading of work in fluid simulation and rendering will be separated in this section for clarity.

2.1 Simulation

2.1.1 Fluid Simulation

The notes of Robert Bridson and Matthias Muller-Fischer are designed for "developers in industry who want to implement or at least understand the state of the art in graphics fluid simulations". [2] The paper covers the basics of 3D fluid flow and the surrounding calculus and vector mathematics. The notes then move on to discuss methods of implementing a water simulation with surface tension and irregular curved boundaries. The later chapters then discuss more advanced techniques such as "adaptive grid methods, real-time-capable algorithms together with the latest technology in hardware acceleration". [2] The contents of the paper are closely related to the simulation element of the project; however, work surrounding rendering topics are not covered here in the same detail.

2.1.2 Real-time simulation of water surface

This paper, authored by Vladimir Belyaev discusses a number of techniques concerning the simulation and rendering of a water surface. The author of this piece deconstructs the problem into three main tasks, simulation of the surface, optical effects and the rendering technique. The

approaches to these problems differ from the approach of this project, although the author does identify refraction and reflection as points of focus, suggesting the solution "for the optic effects simulation is ray-tracing or reverse ray-tracing" [3]. The paper then advances to discussing the mathematics involved in generating the heights for the fluid surface. Evaluating both 2D and 3D mesh solutions presenting the advantages and disadvantages of both options. Due to the paper only concerning the fluid surface, and the problem of this project evaluating both surface and body of the water volume, only some elements of this paper were of direct relevance, although it provided good insight towards the most efficient approaches to particular simulation issues regarding computational speeds.

2.1.3 Simulating Ocean Water

Simulating Ocean Water by Jerry Tessendorf is a paper "intended to give computer graphics programmers and artists an introduction to methods of simulating, animating, and rendering ocean water environments". [4] The paper discusses methods of simulation for fluid dynamics based on the Navier-Stokes equations. Tessendorf suggests the use of Gerstner wave models, and presents advantages and disadvantages of such an approach, then discussing the use of a statistical wave models approach suggesting that the “nice mathematical and statistical properties” of values from decomposition of the wave height fields make it a better solution. This solution makes use of Fast Fourier Transforms (FFT) to accelerate this process making it a quick solution, useful to this project when attempting to generate a real time application.

In addition to this, the paper discusses approaches to rendering the ocean mesh using the generated height field and slope values. Tessondorf suggests radiosity as a possible technique, where experiments presented within the paper are run on a scene containing the water surface, the air, the sun (light source) and the water volume underneath the surface. This relates directly to the scene required to be constructed in this project. The paper also presents solutions to visual elements of the problem, providing solutions to solve for multiple refraction and reflection behaviours between the air-water interface, such as the Frsenel shading method.

2.2 Rendering

2.2.1 Real Time Rendering *(Fourth Edition)*

The basis of this project is using mathematics and programming techniques to formulate the solution. This literature, discusses a broad range of mathematical topics regarding computer

graphics, and most importantly, implementing them to construct a program that responds in real-time, which is extremely relevant to the problem considered for this project. The book explores a range of techniques and provides examples of how they are installed in industry.

The paper discusses how spherical particles can be used as a 2D based solution for "simulating fire, smoke, explosions, water flows, whirling galaxies, and other phenomena". [5] This section contains valuable information relevant to the project as it not only demonstrates an interesting approach to simulate fluid dynamics, but also details how to shade these particles so they can be visible in the render.

The book also explores another technique of rendering, which takes a more 3D approach to simulating fluids.

Although not discussing all topics in absolute depth, the book diverts the reader to great resources to study topics in greater detail.

2.2.2 GPU Gems 3

GPU Gems 3 is the third edition of a 3-book series published by NVIDA Corporation, the world leading computer graphics company. The series "is a collection of state-of-the-art GPU programming examples" [6], containing information on how to utilize their GPUs architecture to its greatest potential and harness their most modern technologies. The literature is constructed so that each chapter explores a particular branch or technique of GPU programming, explaining key mathematical principles involved with implementing the technique and also outlining methods of approaching particular problems. Examples provided also inform the reader on how and where particular techniques are deployed in industry. Topics within this literature range from in-depth approaches of rendering realistic skin and hair to parallel programming techniques related to machine learning.

Modern GPU technologies such as CUDA are incorporated into the discussions and demonstrations in the literature, CUDA is NVIDIA's general purpose GPU platform (GPGPU) which "accelerates applications across a wide range of domains from image processing, to deep learning, numerical analytics and computational science." [1] and GPU Gems demonstrates methods to utilize CUDAs features across these disciplines. Chapters within [6] discuss 3D methods

[1] https://developer.nvidia.com/CUDA-zone

in rendering fluids, discussing equations of motion revolving around the Navier-Stokes equations and also describing various volume rendering techniques. The examples provided in this literature involve the use of older technologies such as DirectX10, where as for this solution will look on implementing this task using modern APIs hoping to harness modern graphical technologies to build on the work done in this literature.

2.2.3 Animation and rendering of complex water surfaces

This paper by Douglas et al. discusses rendering and animation of water at a photo-realistic level. Methods presented in this paper use less approximation to achieve far more visual accuracy. Such algorithms have more suitability in pre-rendered scenarios although the paper provides interesting arguments and physically based rendering techniques that can be considered for the problem of this project.

2.2.4 Advanced illumination techniques for GPU volume raycasting

This article by Marcus Hadwiger et al. describes the rendering method of volume ray casting in detail, including explanations of the surrounding concepts.

> "Experts agree that GPU-based raycasting is the state-of-the art technique for interactive volume rendering" [7]

The literature also explains additional techniques such as "ambient occlusion and simple Monte-Carlo based approaches to global illumination".

Due to this project being concerned with rendering of a fluid volume, the contents of this paper are of close relevance to this project.

2.2.5 Volume Rendering For Games

This piece is a presentation created by Simon Green representing NVIDIA Corporation. The presentation suggests that volume rendering is an adequate solution where real world phemonmena "cannot be easily represented using geometric surfaces" [8], using clouds, smoke and fire as examples. These phenomena fall under the fluid category and make it relevant to this project. The discussion then moves to how volume rendering can be implemented, as well as providing shader code examples for the implementation of volume rendering for a flame. The real-time constraint that exists on video games is what makes the methods described in [8] so relatable to this project.

2.2.6 Acceleration techniques for GPU-based volume rendering

The contents of this article, by J. Kruger et al. cover techniques for improving volume rendering for programs to run at interactive rates. This relates to the real-time demands of this project. The paper discusses methods to "reduce per-fragment operations" [9], introducing techniques such as empty-space skipping and early ray termination. Topics within this literature will help identify areas for optimization for the project.

Chapter 3

Tools

To approach the problem and implement the solution at a high standard, the most suitable tools need to be utilized to assist in producing a responsive and interactive program.

3.1 Base Language

The foundation of the program will be responsible with handling data and used in conjunction with the Graphical API *(See Section 3.2)* to build a framework for the solution. To accomplish this, the language C++ will be used. Games on Windows and console developed by C++ “occupied 75% or more of the market” according to [10]. This is a reliable indicator that this language is suitable for the project due to the large graphical engines modern video games exhibit, that demand output in real-time.

3.1.1 Brief Introduction to C++

"Except for minor details C++ is a superset of the C programming language" [11] and was developed in 1979 by Bjarne Stroustrup.

> "The C++ language features most directly support four programming styles:
>
> - Procedural programming
> - Data abstraction
> - Object-oriented programming
> - Generic programming" [12]

The paradigms and functionality the language offers will prove more than adequate to construct the base of the program.

Despite its age the "game industry is heavily dependent on C and C++" [13], the language provides the programmer low-level access to registers and memory which provides large amounts of control with how implemented programs interact with the hardware it is deployed upon.

In addition to this industry leading Graphical API's *(See Section 3.2)* are largely collaborative with the language, with programs “exhibiting high performance” [14] when used with C++.

3.2 Graphical API

The purpose of a Graphical API is to provide the programmer with an avenue to communicate directly with graphical hardware. Industry leading APIs are OpenGL, DirectX and Vulcan. Vulcan and OpenGL provide support for cross platform however that isn't a requirement of the project. All provide sufficient features to compute modern graphical problems efficiently.

A key element in selecting Graphical API is its effectiveness of running compute shaders *(a shader that is used for calculations rather than for rendering purposes)*, and/or its interoperability with CUDA *(See Section 3.4)*, OpenCL or other general purpose GPU programming frameworks.

Although Vulcan, DirectX and OpenGL all provide CUDA interoperability to some degree, interoperability is very lightly supported for DirectX12. There is a large amount of support for interoperability between OpenGL and CUDA making it a leading candidate for use in the project.

For this project, the focus is more on the mathematical aspect of fluid simulation and the rendering technique itself so it is desirable to keep the code base as concise as possible. OpenGL is far more compact and minimal when compared to its rivals, although DirectX12 and Vulcan delegate far more control to the programmer. However, OpenGL provides more than enough functionality and control to implement the solution effectively and the problem is not large enough to benefit from optimizations available in alternative APIs.

As a result, OpenGL will be used as the Graphical API to conduct the project in conjunction with its assisting libraries.

3.2.1 Brief Introduction to OpenGL

Open Graphics Library is an "industry-standard, cross platform Application Programming Interface (API)" [15] developed and supported by Silicon Graphics Inc. in 1992.

OpenGL is constantly evolving in parallel with modern GPUs with a large cooperate backing and user base. OpenGL comes bundled with a large array of helper functions and libraries which allow for robust and flexible programs to be constructed.

3.3 Shader Language

Graphical programs make use of a conceptual model known as the graphical pipeline. Within the graphical pipeline exist shaders, which are used to manipulate data at particular points within the pipeline. Many of these shaders are programmable, and are configured using shader languages. In this case OpenGL's accompanying shader language GLSL will be used.

3.3.1 Brief Introduction to GLSL

OpenGL Shader Language (GLSL) is OpenGL's accompanying shader language. It was created to give developers greater control over the *graphics pipeline* - its natural link to OpenGL makes it an obvious choice. GLSL has very similar syntax to the C programming language making it a great fit for this C++ based project.

3.4 CUDA

CUDA is a computing platform and API created by NVIDIA Corporation to provide parallel computing capability on their CUDA-enabled GPUs. CUDA is designed to function with languages such as C and C++ making it suitable for use within the project.

CUDA utilizes the parallel capabilities of the GPU, which makes it very effective for large computational problems that can be deconstructed into smaller parallel components to be run across multiple cores; this includes problems in the field of AI, Machine Learning and most applicably Simulations.

CUDA programs are known as ‘kernels’, which will be used to implement the simulation calculations of the fluid.

The target machine for implementation contains a CUDA-enabled GPU enabling the use of the CUDA toolkit to develop the program.

3.5 Other Libraries and APIs

To assemble a complete program it is necessary to include additional libraries to provide advanced data types and mathematics not provided by C++ standard library, as well as tools to help initialize and run the application.

3.5.1 A Brief Introduction to GLM

OpenGL Mathematics is a C++ mathematics library, based on the specifications of GLSL [16]. Within the library are implementations of the most common mathematical elements used within graphics applications, such as matrices and vectors and their associated operations.

3.5.2 A Brief Introduction to GLFW

Graphics Library Framework is a lightweight cross-platform windowing and input handling tool. For the project, this framework will be used to handle input from the user via the keyboard and mouse to provide interactivity within the application.

3.5.3 A Brief Introduction to GLAD

GLAD is an OpenGL loading library, which is used to load OpenGL functions and extensions at run-time

3.6 Target Hardware

The program will be run using a Intel i7-9700k CPU @ 3.60GHz, with 16GB of RAM running Microsoft Windows 10, in addition to this, the target machine hosts an NVIDIA RTX 2080 GPU (Turing), which is CUDA enabled with compute capability of 7.5.

Chapter 4

Approach

In order to implement the problem of simulating and rendering a fluid in real-time, all approaches to such a problem must be considered. Methods discussed in *Related Work (See Section 2)* have highlighted that the field for simulating fluids is extremely broad. Additionally, there are many options available when it comes to rendering, appropriate methods must be decided to ensure accomplishment of the project objectives *(See Section 1.2)*.

4.1 Fluid Model

This section discusses different fluid types that could be modelled to accomplish the project objective. The section also explores mathematical/algorithmic possibilities and limitations with respect to the fluid model selected.

4.1.1 Compressible vs Incompressible Fluids

Compressible fluid flow describes a fluid which sees its volume change throughout time. "Applications of compressible fluid flow theory are in the design of high speed aircraft" [17] and interactions aircraft have on the surrounding air for example a sonic boom. However, fluids don't change their volume commonly and calculations to simulate compressible fluids are expensive [2]. Conversely, the mathematics involved in animating incompressible fluid flow can be quite efficient, and can enable the use of the famous incompressible Navier-Stokes equations.

4.1.2 States of Matter

Fluids are often used to describe liquids, but the term also encapsulates gasses or any substance that deforms when external force is applied. The choice of matter will affect the way in which behaviour can be calculated and its appearance can be rendered.

Liquid, more specifically water, can be categorized further into *Ocean* - exhibiting sea like behaviours such as waves; and *River* - with fast flowing continuous bodies of water.

With gases all aspects of the fluid are considered from a rendering viewpoint, however, with liquids the most visually interesting artifact is the interaction between the liquid-air boundary. This provides room for visual approximations when rendering liquids as the 'surface' of the liquid is the most crucial feature to render correctly rather than the complete fluid. When modelling gasses, values such as temperature and density can be tracked to evaluate appearance. For a fluid such as a fire an additional value of a *'reaction coordinate'* must be stored to track ignition time to alter the appearance of the flame. [6]

These are just a few of the differences that exist between the different fluid phenomena; however it is enough to highlight the effect that the choice of matter has upon the implementation of the solution.

4.1.3 Dimensionality

The method used to look to model the fluid at a conceptual level will effect the mathematics and algorithms involved in simulating and rendering the fluid. This can be deconstructed into 2D or 3D based solutions.

4.1.3.1 2D Techniques

2D approaches to modelling fluids are common within both industry and theory, as seen in *Related Work (See Section 2)*. Popular techniques for modelling fluids in 2D revolve around particle based solutions, where the elements of the fluid or even the molecules themselves are modelled using particles on a 2D domain. The Lattice-Boltzman Method (LBM) can be applied to a 2D space. This method makes use of a particle with fictive particles, containing vectors holding values for velocity. The D2Q9 is a popular model for elements of the lattice (2 dimensions and 9 vectors). There are two main steps that evolve the fluid over time in LBM. The first step is the collision step; which utilizes the Bhatnagar, Gross and Krook (BGK) [18]

model, which considers relaxation to equilibrium due to collisions amongst molecules in the fluid. The second step being the *streaming step* which calculates where elements in the fluid would have moved in the next time step. LBM is designed to run on parallel architectures making it efficient to calculate on a GPU [19].

Expansions on particle based models are available, such as wave packets, which "presents a method for simulating water surface waves as a displacement field on a 2D domain" [20].

4.1.3.2 3D Techniques

3D techniques to modelling and simulating fluids typically involves resolving values of the fluid at particular points to a 3D texture, subdividing the fluid model into cubical cells. "There is a straightforward mapping between grid cells and voxels in 3D textures" [6] on GPUs making it a useful approach. Lattice-Boltzmann methods can also be scaled to 3D, using the lattice vector model of 3D19Q.

4.2 Project Simulation Method

This section describes the method used to simulate the fluid model.

The simulation will be deployed using **water** as the fluid state in an **ocean** model, assumed to be **incompressible**, using a **2D** based approach to the model.

The works of Tessendorf [4] are used to drive our solution, in which the discussed ocean wave model is loosely based on the Naiver-Stokes equations of fluid motion. More specifically the *Statistical Wave Model* discussed making use of Fast Fourier Transform (FFT) to generate our wind driven ocean waves which will be represented as a height field mesh.

4.2.1 Brief Introduction to Ocean Wave Model

In this section I will cover algorithms discussed by Tessendorf [4] that will be used as the body of the ocean simulation.

Tessendorf [4] states that statistical models are favoured within Oceanographic literature. Statistical models are comprised on the ability to decompose the wave height field as a combination of sinusoidal waves with cosine waves. For the case of the surface of the wave model, Inverse-FFT can be used to efficiently perform this decomposition.

Equation 4.1 is used by Tessendorf [4] for the composition of sin and cosine waves.

$$h(x,t) = \sum_k \tilde{h}(k,t)\exp(ik\cdot x) \tag{4.1}$$

Where wave amplitude is denoted as $h(x,t)$ and $\tilde{h}(k,t)$ in spatial and frequency domain respectively. $x = (x,z)$ represents wave position and the vector $k = (k_x,k_z)$ stands for the wave direction. $k_x = \frac{2\pi n}{L_x}, k_z = \frac{2\pi m}{L_z}$, where m and n are integers bound to the ranges $-\frac{N}{2} \leq n < \frac{N}{2}$, $-\frac{M}{2} \leq m < \frac{M}{2}$.

Inverse-FFT can be used to generate the height field for each x for respective values of n and m.

4.2.1.1 Initializing Heightfield

In order to generate $\tilde{h}_0(k)$, Tessendorf [4] suggests the use of a spatial spectrum, namely Phillips Spectrum [21].

$$P_h(k) = A\frac{\exp\left(\frac{-1}{(kL)^2}\right)}{k^4}|\hat{k}\cdot\hat{w}|^2 \tag{4.2}$$

Where L represents the maximum amount of waves generated from a wind speed V, where g is the gravitational constant. A is a numerical constant, and the vector $\hat{w}$ is the wind direction. The cosine factor $|\hat{k}\cdot\hat{w}|^2$ removes waves moving in a perpendicular direction to wind direction.

In the case of a wind driven ocean height field Tessendorf [4] provides the equation 4.3 based on the Phillips Spectrum.

$$\tilde{h}_0(k) = \frac{1}{\sqrt{2}}(\xi_r + i\xi_i)\sqrt{P_h(k)}, \tag{4.3}$$

ξ_r and ξ_i are Gaussian random numbers with mean 0 and standard deviation of 1.

4.2.1.2 Calculating Heightfield

When provided with a dispersion relation $\omega(k)$, wave amplitudes at time t are able to be calculated.

$$\tilde{h}(k,t) = \tilde{h}_0(k)\exp(i\omega(k)t) + \tilde{h}_0^*(-k)\exp(-i\omega(k)t) \tag{4.4}$$

Tessendorf [4] explains that this method is efficient due its reliance on FFTs and that the wave field at any time can be calculated without knowing another wave fields value at another

given time.

It is suggested by Tessendorf [4] that values of N and M for the Fourier grid should be situated between 16 and 2048, in powers of two.

4.3 Project Render Method

The contents of this section discusses the methods we will employ to render the fluid model.

With the steps discussed in *Project Simulation Method (See Section 4.3)* the project will have a functioning simulation, but with no visual representation of the fluid. In order to gain a visual output it is necessary to have a rendering method that accurately displays the state of the liquid but not at a computational cost that will compromise the simulation processing.

To accomplish this, a Blinn-Phong shader will be used, where height and slope values from the wave heightfield are used to calculate surface normals to perform shading.

4.3.1 Brief Introduction to Phong Shading

The Phong shading model is the amalgamation of constant ambient light, diffuse reflection of rough surfaces and reflection of light on shiny surfaces.

4.3.1.1 Ambient Component

Objects within a scene will have an assumed constant ambient color, this constant value will be multiplied by the color omitted from the light source to provide the objects base ambient color.

4.3.1.2 Diffuse Component

Diffuse lighting is built on the idea that lighting intensity of an object is related to how close its surface is aligned to light rays outputted from the light source. This is done by omitting a light ray from the light source towards an object, then using the normal of that object surface point and calculating an ‘angle of incidence’ between the two involved vectors.

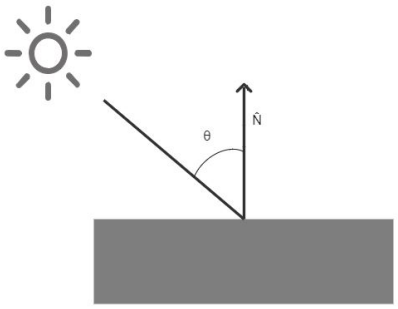


Figure 4.1: Diffuse Lighting Model

4.3.1.3 Specular Component

The specular light component considers that light reflected from a shiny surface is dependant on the angle between the reflected light ray and the view vector. In this model, light is reflected about the surface normal of that the original light ray hits. The intensity of the light reflected can be scaled using a specular factor determining how shiny the modelled object is. [22]

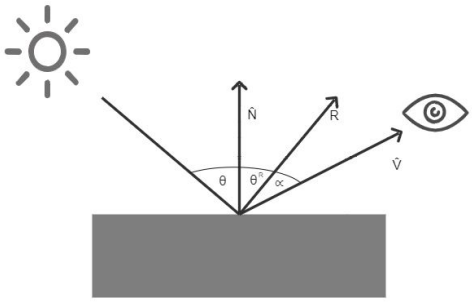


Figure 4.2: Specular Lighting Model

4.3.2 Brief Introduction Blinn-Phong Shading

Blinn-Phong shading is an adaptation by James Blinn to the Phong shading model proposed in 1977 [23]. The specular component of the Phong shading breaks when the view and reflection vector is larger than 90 degrees. Blinn's extension of Phong shading introduces a 'halfway vector' which is calculated as halfway between the reflected ray vector and the view vector, the angular distance between the halfway vector and surface normal will never be in excess of 90 degrees, allowing for the shader to always produce a correct specular component output. This

model is what will be employed for the shader. The halfway vector for Blinn-Phong shading calculated by:

$$H = \frac{L+V}{\|L+V\|} \tag{4.5}$$

Where L is the light direction vector, and V is the view direction vector.

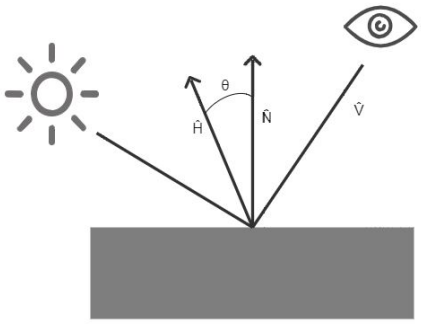


Figure 4.3: Blinn-Phong Shading Model

4.4 Project Framework

Before implementing the approach to the solution, considerations must be made on how to utilize the available tools to generate the best results. It is important that the program is setup in a memory efficient manner, and that appropriate code is executed in parallel where possible to improve speeds of the program.

4.4.1 Simulation Kernels

In order to implement the simulation, CUDA will be used to develop 'kernels' that will calculate parallel elements of the simulation and FFTs. The alternate option to this approach would be to use 'compute shaders' written in GLSL. The advantage of using CUDA rather than compute shaders is that CUDA's toolkit is equipped with an abundance of efficient support mathematical libraries such as cuFFT (CUDA Fast Fourier Transforms) which will be used within the program.

Kernels will be executed in 8x8x1 grid blocks, where a grid of CUDA threads is scaled based on the dimensions of the mesh size.

4.4.2 Shaders

To visualize the results from the simulation GLSL shaders will be used. It will only be necessary to have two programmable shaders per pipeline, the Vertex Shader, to perform per vertex operations necessary and a Fragment Shader to perform per pixel operations.

4.4.3 Device Memory

Within the program, memory stored on the GPU should aim to be accessible by both CUDA and OpenGL without duplication. CUDA and OpenGL interoperability will be used to ensure that shaders and simulation kernels have access to the same memory locations and data buffers.

4.4.4 Camera

For the program, it would be useful if the user is able to move around the ocean mesh to gain different visual perspectives of the model, and to also provide variable view angles for the Blinn-Phong shader. It will be necessary that the program is able to receive input from peripherals available to the target machine, such as a keyboard and/or mouse.

4.4.5 Measurements

A feature to measure the efficiency and speeds of the rendered simulation needs to exist within the system. To do this, frame time will be measured and displayed in milliseconds (ms) which will measure the time taken to render each frame, and to also provide value in frames per second, to gain better insight to what speeds are being attained.

4.5 Project Plan

Before implementing the program, it is necessary that the design process of the project is planned, to ensure completion of the project within the provided time to an acceptable standard.

4.5.1 Design Methodologies

To implement the program a top-down design approach will be used. This is to segregate components of the problem into smaller elements which are easier to develop. When a program element is completed it is merged with 'sister' elements that belong to the same parent, this is performed for each sub element in the program until the solution is complete. As for the methodology used for the whole project, the waterfall [24] approach will be used. Where information is gathered on the related topic, the system is designed, implemented and then tested. This rigid design structure is applicable as the end goal of the project is already known, so it is intuitive to work through the project in an organized structured manner, where there will be clean transitions at the end of each development phase.

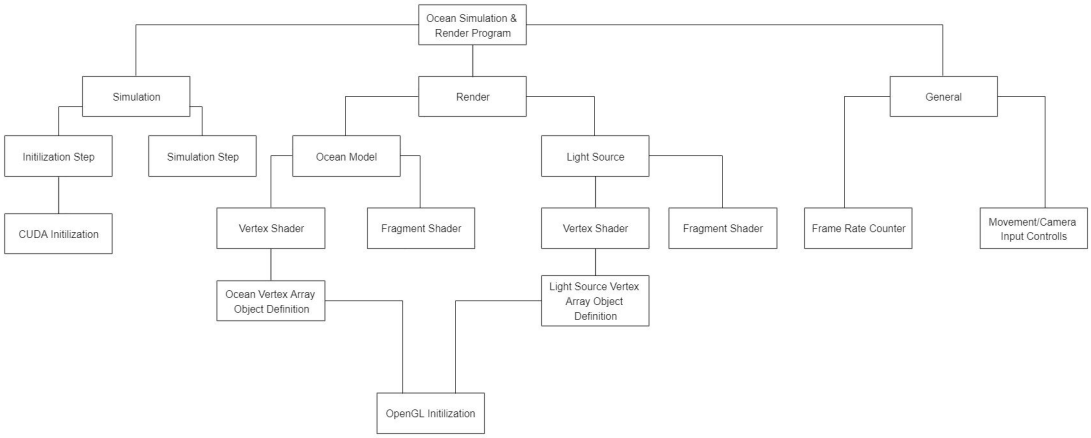


Figure 4.4: Top Down Design Model of System

Chapter 5

Implementation

The following sections within this chapter detail how key components of the program were implemented, following the methods discussed in the Approach (See Chapter 4).

5.1 Simulation

This section describes how core elements of the Ocean Model were simulated.

5.1.1 Initializing Heightfield

Following methods of Tessendorf [4] as discussed in the approach *(See Section 4.2.1.1)*, initialization of the height field is required. Initialization of the height field for t=0, is only performed once, for this reason this was implemented on the CPU.

```
1  /**
2   * Initialize heightfield for t=0.
3   *
4   * (Using Jerry Tessendorfs algorithm, with Phillips specturm)
5   *
6   */
7  void initHeightfield(float2* heightfield_0)
8  {
9    for (unsigned int y = 0; y <= MESH_WIDTH; y++)
10   {
11     for (unsigned int x = 0; x <= MESH_HEIGHT; x++)
12     {
13       float2 k; //Wave direction vector.
14       k.x = (-(int)MESH_WIDTH / 2.0f + x) * (2.0f * PI_FLOAT / PATCH_WIDTH);
```

```
15        k.y = (-(int)MESH_HEIGHT / 2.0f + y) * (2.0f * PI_FLOAT / PATCH_HEIGHT);
16
17        float Ph = sqrtf(phillipsSpectrum(k.x, k.y, windDirection, V, A)); //Square
              root our phillips spectrum.
18
19        if (k.x == 0.0f && k.y == 0.0f)
20          Ph = 0.0f;
21
22        float Xi_r = gaussianRandom();
23        float Xi_i = gaussianRandom();
24
25        float heightfield_0_r = Xi_r * Ph * SQUARE_ROOT_HALF_FLOAT;   //Real
              Component
26        float heightfield_0_i = Xi_i * Ph * SQUARE_ROOT_HALF_FLOAT;   //Imaginary
              Component
27
28        int i = y * SPEC_WIDTH + x;
29        heightfield_0[i].x = heightfield_0_r;
30        heightfield_0[i].y = heightfield_0_i;
31      }
32    }
33  }
```

Listing 5.1: Implementation of h0 (Initialize Heightfield for t=0)

5.1.2 Calculating Heightfield

Now there is an initialized heightfield, Equation 4.4 can be implemented to calculate values for time t, which will run as time elapses in the program. These calculations are ran across each point in the mesh so it will be implemented using CUDA so that it can be computed in parallel.

```
1  //Generate wave heightfield for time t using dispertion relationship from initial
       heightfield.
2  __global__ void calculateHeightfield(float2 *heightfield_0, float2 *heightfield_t,
        unsigned int specWidth, float t, float patchWidth, float patchHeight,
       unsigned int meshWidth, unsigned int meshHeight)
3  {
4
5      unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
6      unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
7
8      unsigned int input_i = y*specWidth+x;
```

```
9       unsigned int input_i_mirrored = (meshHeight - y)*specWidth + (meshWidth - x);
            // Mirrored index
10      unsigned int output_i = y*meshWidth+x;
11
12      //Calculate wave direction vector.
13      float2 k;
14      k.x = (-(int)meshWidth / 2.0f + x) * (2.0f * PI_FLOAT / patchWidth);
15      k.y = (-(int)meshWidth / 2.0f + y) * (2.0f * PI_FLOAT / patchHeight);
16
17      // Calculate dispersion w(k) using inital values of heightfield.
18      float k_len = sqrtf(k.x*k.x + k.y*k.y);
19      float w = sqrtf(gravity * k_len);
20
21      if ((x < meshWidth) && (y < meshHeight))
22      {
23          float2 heightfield_0_k = heightfield_0[input_i];
24          float2 heightfield_0_mirrored_k = heightfield_0[input_i_mirrored];
25
26          heightfield_t[output_i] = cplxAdd(cplxMult(heightfield_0_k, cplxExp(w * t)
                ), cplxMult(cplxConj(heightfield_0_mirrored_k), cplxExp(-w * t))); //
                Frequency Space Output
27      }
28  }
```

Listing 5.2: Implementation of Heightfield Calculation (Calculate Heightfield for time t)

The values in the heightfield for time t are generated by using the dispersion relationship between the initial heightfield values.

5.1.3 Generating Heightmap

In order for the rendering engine to visualize the values of the heightfield, amplitude values need to be stored into a height map that is passed to the shaders.

The input to this function is taken from the output of the Fast Fourier Transform with corrected polarity (sign) on heightfield values; this is executed for each point in the mesh, and therefore is required to be implemented as a CUDA kernel to be computed in parallel.

```
1   //Update values of height map using output from cuFFT.
2   __global__ void generateHeightmap(float  *heightmap,
3                                     float2 *heightfield_t,
4                                     unsigned int width)
5   {
6       unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
7       unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
```

```
8       unsigned int i = y*width+x;
9
10    float polarity;
11    if ((x + y) % 2 == 1)
12    {
13      polarity = -1.0f;
14    }
15    else
16    {
17      polarity = 1.0f;
18    }
19
20      heightmap[i] = heightfield_t[i].x * polarity;
21  }
```

Listing 5.3: Generation of Heightmap

5.1.4 Calculating Slope

The slope between points in the heightfield are also needed to calculate surface normals for the rendering engine. This is done by calculating partial differences between points on the heightmap, and due to this being calculated for each point set it is required to be executed in parallel, and is implemented as a CUDA kernel.

```
1  // Calculate slope between two points by using partial differences.
2  __global__ void calculateSlope(float *heightmap, unsigned int width, unsigned int
       height, float2* outputSlope)
3  {
4      unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
5      unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
6      unsigned int i = y*width+x;
7
8      float2 slope = make_float2(0.0f, 0.0f);
9
10     if ((x > 0) && (y > 0) && (x < width-1) && (y < height-1))
11     {
12         slope.x = heightmap[i+1] - heightmap[i-1];
13         slope.y = heightmap[i+width] - heightmap[i-width];
14     }
15
16     outputSlope[i] = slope;
17 }
```

Listing 5.4: Calculating Slope

5.1.5 Initializing Simulation

Before the whole simulation method can be called, it is required to initialize CUDA and the simulation itself, using the method discussed earlier in *Initializing Heightfield (See Section 5.1.1)*. It is also required that the memory buffers in device (GPU) memory are created and synchronised with OpenGL using CUDA-OpenGL interoperability. This will ensure that both OpenGL and CUDA share device memory, reducing the amount of space taken by the simulation data, and not requiring device-device memory copies - which can introduce large overhead if called on each iteration of the game loop. Implementation of this is present in the appendix of the document in section A.1.

5.1.6 Simulation Method

With these elements in place, they can be assembled to produce the complete simulation pipeline. Once the heightfield is initialized, it is necessary to regenerate the heightfield in the frequency domain, then inverse-FFT is used to convert the heightfield from frequency domain to spatial domain so that the height map and slope values can be updated ready for the next shader run.

```
1  void simulation()
2  {
3    size_t size_bytes;
4
5    //Regenerate wave heightfield (In Frequency Domain)
6      cu_CalculateHeightfield(device_heightfield_0, device_heightfield_t, SPEC_WIDTH
           , anim_time, PATCH_WIDTH, PATCH_HEIGHT, MESH_WIDTH, MESH_HEIGHT);
7
8      //Peform inverse-FFT to convert to convert heightfield to spatial domain
9      checkCudaErrors(cufftExecC2C(fftPlan, device_heightfield_t,
           device_heightfield_t, CUFFT_INVERSE));
10
11     //Update Heightmap values
12     checkCudaErrors(cudaGraphicsMapResources(1, &cu_heightVB_res, 0));
13     checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&
           device_heightmap_ptr, &size_bytes, cu_heightVB_res));
14
15     cu_UpdateHeightMap(device_heightmap_ptr, device_heightfield_t, MESH_WIDTH,
           MESH_HEIGHT);
16
17     //Calculate Slope values
18     checkCudaErrors(cudaGraphicsMapResources(1, &cu_slopeVB_res, 0));
19     checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&
           device_slope_ptr, &size_bytes, cu_slopeVB_res));
```

```
20
21      cu_CalculateSlope(device_heightmap_ptr, MESH_WIDTH, MESH_HEIGHT,
            device_slope_ptr);
22
23    checkCudaErrors(cudaGraphicsUnmapResources(1, &cu_heightVB_res, 0));
24      checkCudaErrors(cudaGraphicsUnmapResources(1, &cu_slopeVB_res, 0));
25
26  }
```

Listing 5.5: Simulation Method

5.2 Rendering

For the renderer, two main shaders will be used; a vertex shader, and a fragment shader. The vertex shader will receive vertex input from the mesh, as well as the heightmap and slope values calculated in the simulation. The vertex shader code generates surface normals from the heightmap and slope values output from the simulation kernel; the code listing of the vertex shader is available under section A.5 in the document appendix, but is omitted from this section as it is not related to the Blinn-Phong Shader model. The fragment shader implements a simple Blinn-Phong shader, in which light calculations are made from the point light source in the scene.

```
1  #version 460 core
2
3  out vec4 fs_Colour;
4
5  struct Light{
6      vec3 pos;
7      vec3 ambient;
8      vec3 diffuse;
9      vec3 specular;
10 };
11
12 struct Ocean {
13     vec3 ambient;
14     vec3 diffuse;
15     vec3 specular;
16     float shininess;
17 };
18
19
20 in VERTEX_SHADER_OUTPUT
21 {
```

```
22      vec3 normal;
23      vec3 fragPos;
24
25  } fs_In;
26
27
28  uniform vec3 viewPos;
29
30  uniform Ocean ocean;
31  uniform Light light;
32
33  void main()
34  {
35
36      //Ambient component.
37      vec3 ambient = ocean.ambient * light.ambient;
38
39      //Diffuse component.
40      vec3 norm = normalize(fs_In.normal);
41      vec3 lightDir = normalize(light.pos - fs_In.fragPos);
42
43      float diff = max(dot(norm, lightDir), 0.0);
44      vec3 diffuse = light.diffuse * (diff * ocean.diffuse);
45
46      //Specular component.
47      float spec = 0.0;
48
49      vec3 viewDir = normalize(viewPos - fs_In.fragPos);
50      vec3 halfDir = normalize(lightDir + viewDir);
51      spec = pow(max(dot(norm, halfDir), 0.0), ocean.shininess * 2); //Double
            shininess to get results closer to regular phong.
52      vec3 specular = (spec * ocean.specular) * light.specular;
53
54      //Combination
55      vec3 result = ambient + diffuse + specular;
56      fs_Colour = vec4(result, 1.0);
57  }
```

Listing 5.6: Ocean Fragment Shader

The Blinn-Phong shader calculates lighting using the ambient, diffuse, specular and shininess factor of the ocean model, implementing the methodology discussed in the Approach *(See Section 4.3.2)*.

5.3 Game Loop

Iteration of real-time logic within an interactive application is often referred to as a ‘game loop’. In this case, the game loop will iterate through the simulation (if animation is on) and the renderer, this ensures that there is an up to date image output to the display where the simulation is constantly being run and simulation values are constantly being redrawn to the screen. Implementation of this game loop is present in A.3 within the document appendix.

5.4 General

This section lightly discusses the implementation of general features in the project, although these features are not related to real-time simulation and rendering, they are necessary to assist in the quality and interactivity of the project.

5.4.1 Camera

In order to place the user in the scene, the camera/eye is what will be used to refer to the users viewpoint. To fully test the deployed lighting algorithm from multiple angles and perspectives in real-time it is required to enable the camera to have its position translated along the scene. As well as this the user must be able to move the cameras ‘tilt’, similar to that of a traditional First Person Shooter game. This was implemented using positional and angular translation. GLFW provided the necessary tools to enable us to receive keyboard and mouse input to manipulate camera values whilst the program runs.

5.4.1.1 Directional Movement (Keyboard Input)

Directional movement in the program is achieved using the keyboard input W, A , S and D to move Up, Left, Down and Right respectively. Input is checked by the system at the end of each render run. Different hardware will execute the render at different speeds, meaning input is being checked at different rates across different machines. Although this is not a critical problem due to this being on offline application, it is an undesired effect, and a very powerful machine could exhibit very fast movement speeds in the program. To avert this, “delta time” is measured, which is the difference in time between the current and last frame, this value is then multiplied by a constant desired movement speed factor to produce a consistent movement

speed. Implementation on what keys were polled for input is presented in A.4, which is located in the document appendix.

5.4.1.2 Angular Movement (Mouse Input)

For the program to provide the user the effect of looking around the world rotations must be applied to view vectors. The Euler angles pitch and yaw are adjusted by measuring offsets in mouse position along the y and x axis respectively.

5.4.2 Scene

To test the shader in different environments the feature of variable scenes was implemented, which have varying light color, scene background color and water color values. Ambient, diffuse and specular color values are able to be toggled using keyboard input, with the user switching between preset scenes. Implementation of the scene presets can be seen in A.2 of the documents appendix.

Chapter 6

Results

This chapter observes and discusses the results, both statistical and visual from the constructed project. Results were gathered to gain information on the visual quality of the render, but also to measure that the program is running in real-time to gather information for evaluation on whether the project goals were achieved.

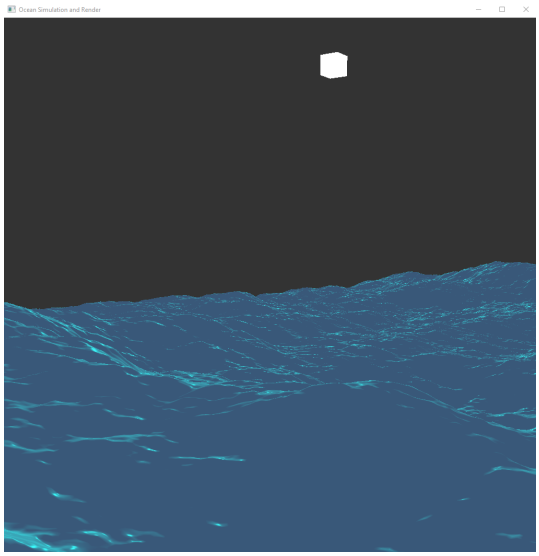
6.1 Real-Time Simulation and Render Test

For this section, two experiments were setup using the constructed program, the tests are setup with variable mesh sizes to test the strain on the computers resources, and to also observe whether visual accuracy varies as a result of this.

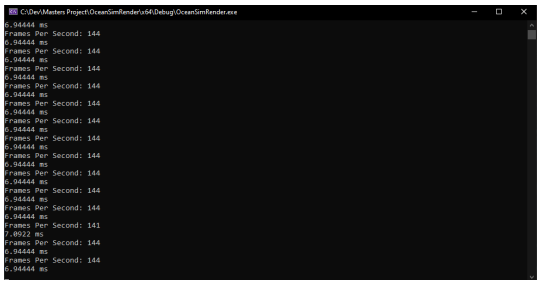
The fail condition for the visual accuracy is if the fluid model is indistinguishable as an ocean-like scene. The condition for failure on frames per second (FPS) is if the program runs at a frame rate less than 60fps, a modern requirement for interactive applications of this scale [25].

The conditions of each experiment are listed as follows:

- Experiment 1: Mesh Size = 1024x1024, Scene = Day, Wind Speed = 1000
- Experiment 2: Mesh Size = 2048x2048, Scene = Day, Wind Speed = 1000

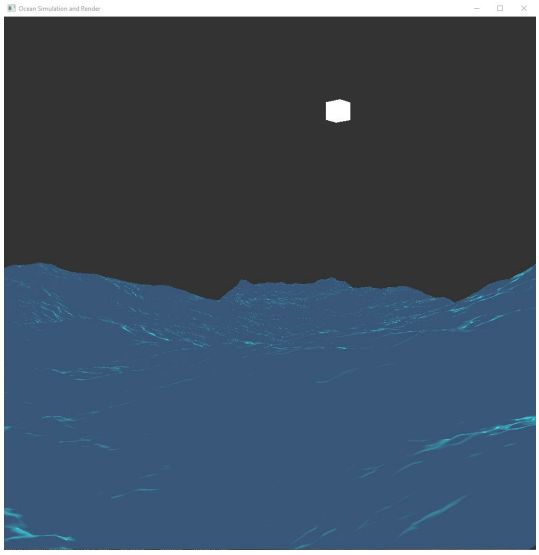


A. Visual Output (Experiment 1)

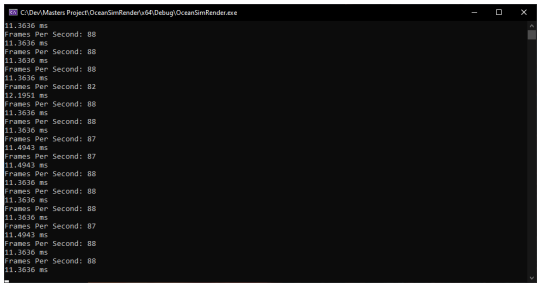


B. Refresh Speeds / FPS (Experiment 1)

Figure 6.1:



A. Visual Output (Experiment 2)



B. Refresh Speeds / FPS (Experiment 2)

Figure 6.2:

As is observable from the results presented from experiment 1 *(See Figure 6.1)* and experiment 2 *(See Figure 6.2)* the program was successful in being able to render the mesh and produce ocean-like fluid.

Experiment 1 was able to render the ocean mesh at frame at the hardware maximum frame rates (144 frames per second on target monitor). This shows that the simulation is able to run and be rendered efficiently, with surplus available computational resources.

Experiment 2 performed rendering and simulation at frames rates of about 88 frames per second. This value is above the failure threshold, although it is apparent that there is now stress on the target machine in computing the simulation and rendering the result. Despite a slower computation time, there appears to be no observable detriment in the quality of the render although the smoothness of the wave simulation is of slightly lesser quality due to the lower frame rate.

6.2 Survey Testing

This section documents testing done which saw examples distributed to a small audience of fifty students and video game enthusiasts who have familiarity with modern graphical quality. The test subjects will be asked several questions on a range of topics with an overall goal to discover how the simulation and render perform aesthetically.

6.2.1 Categorization Test

This test was conducted to validate that the program is able to produce a realistic ocean model. For this test the clients were presented with *Experiment 1 (See Figure 6.1)* from the previous test and asked what category they felt the fluid model belonged to.

The results of the test are depicted below:

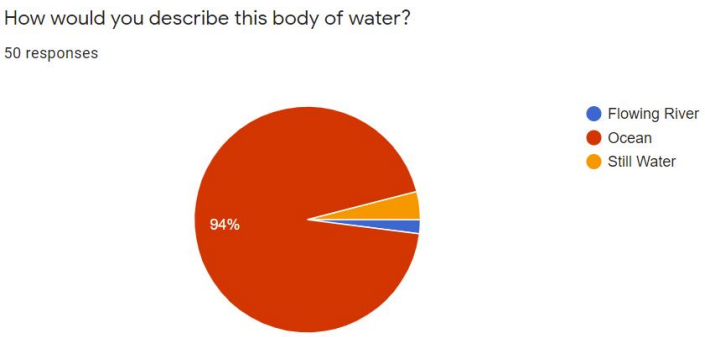


Figure 6.3: Results of Category Survey

The results display that the fluid model was unanimously categorised as an ocean, with 94% of test clients claiming that the body of water was describable as an ocean. This would suggest that the program is more than adequate for simulating and rendering a fluid model.

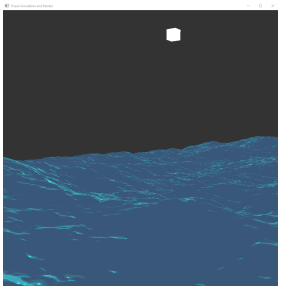
6.2.2 Scene Test

This test is to evaluate the flexibility of the shader program by questioning whether the ocean model is able to be presented in a number of different art styles. This test series will enable the assessment of which scenes are most adequate to model their respective scenarios.

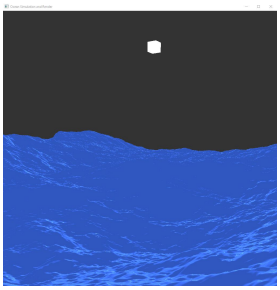
The visual output of the test scenes were distributed to the users and they were asked how satisfied they would be if the model was used to represent ocean water in a modern video game for a particular scene/art style.

The conditions of each experiment are listed as follows:

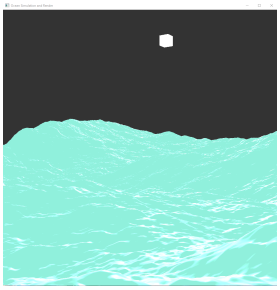
- Scene 1: Day (Realistic)
- Scene 2: Day (Arcade)
- Scene 3: Tropical



A. Day (Realistic)



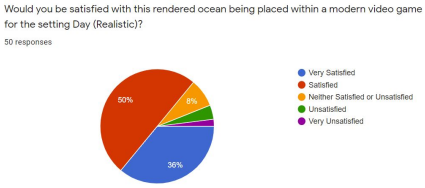
B. Day (Arcade)



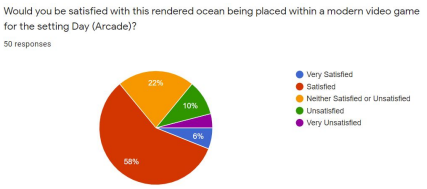
C. Tropical

Figure 6.4: Visual output from 3 Scene Renders (Day (Realistic), Day (Arcade), Tropical)

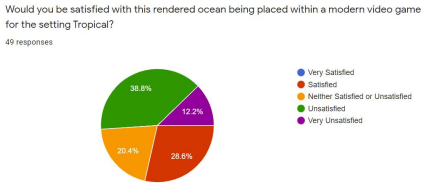
Presented below are the results from the survey:



A. Results of Experiment 1: Day (Realistic)



B. Results of Experiment 2: Day (Arcade)



C. Results of Experiment 3: Tropical

Figure 6.5: Test Results from Scene Survey

As can be seen from the results, the *Day (Realistic)* scene was most effective at pleasing the audience. 86% of the feedback was that the Scene 1 render was either satisfactory or very satisfactory. Allowing the discovery that the colour values in Scene 1 are appropriate for

outputting a realistic looking render of an ocean.

The results for *Day (Arcade)* were varied, with as many as 58% of test subjects claiming they would be satisfied with the ocean model in the relative scenario; however 10% of users stated they would be unsatisfied, and 22% of people not convinced that the model is 'satisfactory or unsatisfactory'. This shows that the values used in Scene 2 are not unanimously convincing to produce visually accurate arcade style water renders; however enough test subjects were satisfied to suggest the render is flexible enough to support different themes in the scene.

Viewing the results from the *Tropical* scene, it is seen that the majority of the feedback was negative. 40% of people were unsatisfied or very unsatisfied with the visual accuracy of the tropical ocean model as <13% of participants were satisfied or better with the output. This is the largest majority of opinion in this result base, allowing us to determine that the tropical scene is inadequate at outputting visually satisfying results.

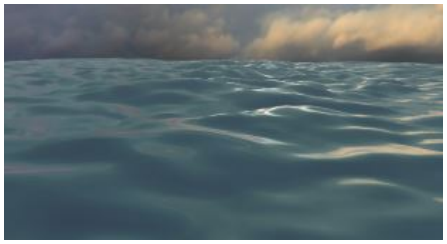
6.2.3 Comparison Test

This test will draw comparison to existing solutions to similar problems to determine whether the project is producing relatively accurate visual results. This will help determine if the implementation is visually competitive with published solutions, and validate that the model is a representation of an ocean.

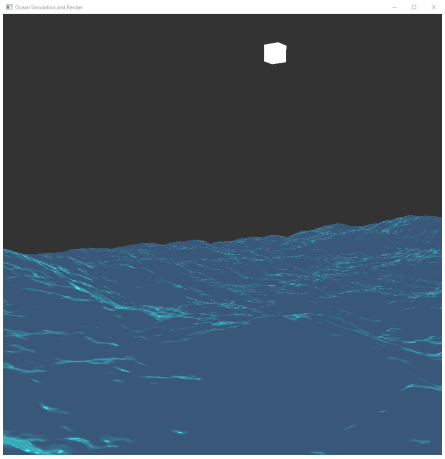
Using the same test pool as the previous two surveys, subjects were presented with three different renders without labels and had to determine which render was the most visually realistic.

The ocean models in question are listed below:

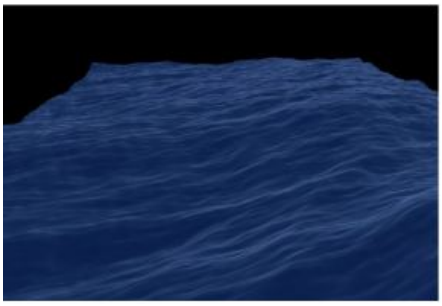
- Ocean 1: Mitchell's Ocean Model [26]
- Ocean 2: This Project Ocean Model (Mesh Size = 1024x1024, Scene = Day (Realistic))
- Ocean 3: Tessendorf's Ocean Model [4]



A. Ocean 1 (Mitchell's Ocean Model [26])



B. Ocean 2 (This Project Ocean Model)



C. Ocean 3 (Tessendorf's Ocean Model

Figure 6.6: Ocean Models Used in Comparison Test

The results of the survey are presented below:

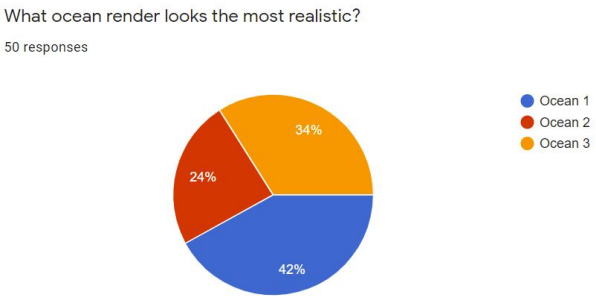


Figure 6.7: Results of Comparison Survey

From the results it can be seen that Mitchell's model [26] appeared the most realistic to the audience. This ocean model was seen to be more realistic than Tessendorf's [4] results by 10% of people. Although, it is important to consider that the implementation of this project makes use of more modern graphical programming technologies when compared to the older

technologies used in the other ocean models . Although not selected as the most realistic render out of the test suite, this projects implementation was considered to be relatively competitive to the other ocean models, with variance in feedback only being within +/- 10% range.

6.3 Evaluation and Further Discussion

From the results gathered in this chapter it is visible that an ocean fluid model was successfully simulated and rendered. The project passed the assessment of the simulation performing at interactive real-time frame rates, which was a requirement for the implementation. From the survey results it can be determined that the fluid render is distinguishable as an ocean, and is of a quality comparable to similar existing solutions when concerning appearance.

Generally speaking, it can be evaluated from the results that it is possible to simulate and render the ocean model with large amounts of computational power being left in reserve. This is desirable, and suggests that this projects' model can be placed into a larger application with little to no detriment to the performance of the application.

With the maximum frame rates attained in the *Real-Time Simulation and Render Test (See Section 6.1)* and when observing the visual quality in Mitchell's ocean render [26] it could suggest that a more intensive render model could be implemented to produce visual results of greater accuracy.

A key problem was encountered within the project the longer it ran; as the duration of run-time increased, the speed of the simulation and the rate of polled input would increase. Although the cause of this behaviour was undiscovered, it can be assumed that this is a result of the simulation stabilizing over time, and that less computational power is required as time evolves.

Chapter 7

Conclusions and Future Work

7.1 Overview

During this project the very large field of Fluid Simulation and Rendering has been briefly explored. With the model implemented (Ocean Simulation) it was learned that Tessendorf' methods [4] are effective when implemented using modern programming tools, as the highly parallel algorithms can be effectively executed on GPUs. It was also identified that Blinn-Phong shading produces semi-realistic results, although greater visual accuracy can be achieved using alternative shading approaches.

The project was able to satisfy the initial set goals *(See Section 1.2)*. The project goals are listed below, with brief justification on how each goal was attained.

- **Implement a program capable of efficiently simulating a fluid**

  As showcased in the results section, the program implemented an ocean fluid simulation, and was able to perform at real-time interactive frame rates.

- **Implement a program capable of rendering a fluid**

  As presented in the results, the combination of the ocean simulation and the Blinn-Phong rendering technique was able to produce a visual representation of an ocean fluid model.

- **Analyse the implemented solution and compare with similar solutions**

  The results section drew comparison between the visual output and the visual quality of similar solutions, where it was found that the implemented solution was comparable to existing solutions.

- **Identify areas of the solution that could be improved and/or optimized**

  Evaluation of the results concluded that use of the Blinn-Phong as a shading model could be improved to account for reflective and transmission phenomena present in ocean water such as the Fresnel model.

7.2 Future Work

Although this project succeeded in satisfying the initial project goals, it was learned there are large revisions that can be made to improve the quality of the simulation and the render. In the future, the program can be amended to facilitate for the stabilization of the simulation. Similar to the "delta time" approach enforced to regulate movement speed between render runs, a refresh delay should be added to ensure that the program runs smooth consistently.

To broaden the challenge presented, insertion of an object such as a buoy or a boat could be made to the ocean model - done by using values in the heightmap to translate the object along the ocean surface. This would present an interesting challenge and an expansion on the original problem.

Concerning the render quality of the program, the solution could be improved using a highly detailed rendering algorithm, such as a ray tracer using Monte-Carlo path-tracing [27]. The efficiency of the simulation results in a large computational power reserve which can be applied to generate more realistic output.

Only the surface of fluid rendering and simulation was covered in this paper both literally and metaphorically; more complex 3D approaches can be considered in future, such as work shown in [2] implementing alternative fluid phenomena such as gas, or fire.

Bibliography

[1] N. H. Tomas Akennie-Moller, Eric Haines, *Real Time Rendering, Thrid Edition*. CRC Press, 2008.

[2] M. M.-F. Robert Bridson, "Fluid simiulation," 2007.

[3] V. Belyaev, "Real-time simulation of water surface," 2003.

[4] J. Tessendorf, "Simulating ocean water," 2004.

[5] N. H. e. a. Tomas Akennie-Moller, Eric Haines, *Real Time Rendering, Fourth Edition*. CRC Press, 2018.

[6] M. Bunnell, *Dynamic Ambient Occlusion and Indirect Lighting*. Nvidia Corporation, 2005.

[7] M. H. et al, "Advanced illumination techniques for gpu volume raycasting," pp. 1–166, 2008.

[8] S. Green, "Volume rendering for games volume rendering for games," 2005.

[9] J. K. R. Westermann, "Acceleration techniques for gpu-based volume rendering," 2003.

[10] S.-Y. R. Jung-Won Byun, "A study on common module modeling method of game software based on directx/c++," 2009.

[11] B. Stoustrup, *The C++ Programming Language, First Edition*. Addison-Wesley, 1985.

[12] ——, *The C++ Programming Language*. Addison-Wesley, 1985.

[13] J. E. S Leutenegger, "A games first approach to teaching introductory programming," pp. 115–118, 2007.

[14] J. L. C. V. Scott Gordon, *Computer Graphics Programming in OpenGL with C++*. Mercury Learning Information, 2018.

[15] R. J. R. et al., *OpenGL Shader Language*, 2004, vol. 3.

[16] (2020) Opengl mathematics. [Online]. Available: https://glm.g-truc.net/0.9.9/index.html

[17] W. E. C. Patrick H. Oosthuizen, *Introduction to Compressible Fluid Flow*. CRC Press, 2013.

[18] M. K. PL Bhatnagar, EP Gross, "A model for collision processes in gases. i. small amplitude processes in charged and neutral one-component systems," 1954.

[19] E. C. et al., "Massively parallel lattice–boltzmann codes on large gpu clusters," 2016.

[20] C. W. Stefan Jeschke, "Water wave packets," 2017.

[21] O. M. Phillips, "The equilibrium range in the spectrum of wind generated waves," 1958.

[22] U. Claussen, "Real time phong shading," 1992.

[23] T. L. Linus Kallberg, "Optimized phong and blinn-phong glossy highlights," vol. 3, no. 3, pp. 1–6, 2014.

[24] Wilfred Van Casteren, "The waterfall model and agile methodologies :a comparison by project characteristics," 2017.

[25] R. Hagström, "Frames that matter," pp. 9–11, 2015.

[26] J. L. Mitchel, "Real-time synthesis and rendering of ocean water," 2005.

[27] J. Hammersley, "Monte carlo methods," 1964.

Appendix A

Implementation Code Listings

A.1 CUDA/Simulation Initialization

```
/**
  * Initialize CUDA, create shared memory buffers
  * and create VBO's create VAO to be passed to our shader program.
  *
  */
void initCuda(int argc, char** argv)
{
  findCudaDevice(argc, (const char**)argv);

  int specSize = SPEC_WIDTH * SPEC_HEIGHT * sizeof(float2);
  int outSize = MESH_WIDTH * MESH_HEIGHT * sizeof(float2);

  // Create CUDA FFT Plan
  checkCudaErrors(cufftPlan2d(&fftPlan, MESH_WIDTH, MESH_HEIGHT, CUFFT_C2C));

  // Allocate device (GPU) memory for our specturm

  checkCudaErrors(cudaMalloc((void**)&device_heightfield_0, specSize));
  host_heightfield_0 = (float2*)malloc(specSize);
  initHeightfield(host_heightfield_0);
  checkCudaErrors(cudaMemcpy(device_heightfield_0, host_heightfield_0, specSize,
      cudaMemcpyHostToDevice));

  checkCudaErrors(cudaMalloc((void**)& device_heightfield_t, outSize));
  checkCudaErrors(cudaMalloc((void**)& device_slope_t, outSize));

  // Create Vertex Buffers for slope and heightmap
  initVBO(&heightVBO, MESH_WIDTH * MESH_HEIGHT * sizeof(float), GL_DYNAMIC_DRAW);
```

```
28    checkCudaErrors(cudaGraphicsGLRegisterBuffer(&cu_heightVB_res, heightVBO,
          cudaGraphicsMapFlagsWriteDiscard));
29
30    initVBO(&slopeVBO, outSize, GL_DYNAMIC_DRAW);
31    checkCudaErrors(cudaGraphicsGLRegisterBuffer(&cu_slopeVB_res, slopeVBO,
          cudaGraphicsMapFlagsWriteDiscard));
32
33    // Create Mesh VBO and Index Buffer.
34    initPosVBO(&posVBO, MESH_WIDTH, MESH_HEIGHT);
35    initMeshIB(&meshIB, MESH_WIDTH, MESH_HEIGHT);
36
37    //Create Verterx array for our buffer objects to be interleaved to be passed to
          our shader.
38    glGenVertexArrays(1, &oceanVAO);
39
40    simulation();
41  }
```

Listing A.1: Initialization of CUDA and Simulation

A.2 Scene Definitions

```
1  //----TROPICAL SCENE----
2  glm::vec3 TROPICAL_AMBIENT(0.56470f, 0.94117f, 0.8627f);
3  glm::vec3 TROPICAL_DIFFUSE(0.56470f, 0.94117f, 0.8627f);
4  glm::vec3 TROPICAL_SPECULAR(0.56470f, 0.94117f, 0.8627f);
5  glm::vec1 TROPICAL_SHININESS(0.2f);
6  glm::vec4 TROPICAL_CLEAR(0.0f, 0.8f, 0.933f, 1.0f);
7
8  //----DAY (Realistic) SCENE----
9  glm::vec3 DAY_REALISTIC_AMBIENT(0.223529f, 0.34509f, 0.47450f);
10 glm::vec3 DAY_REALISTIC_DIFFUSE(0.0f, 0.5f, 0.4f);
11 glm::vec3 DAY_REALISTIC_SPECULAR(0.0f, 0.5f, 0.4f);
12 glm::vec1 DAY_REALISTIC_SHININESS(0.8f);
13 glm::vec4 DAY_REALISTIC_CLEAR(0.0f, 0.8f, 0.933f, 1.0f);
14
15 //----DAY (Arcade) SCENE----
16 glm::vec3 DAY_ARCADE_AMBIENT(0.1882f, 0.3372f, 0.749019f);
17 glm::vec3 DAY_ARCADE_DIFFUSE(0.1882f, 0.3372f, 0.749019f);
18 glm::vec3 DAY_ARCADE_SPECULAR(0.1882f, 0.3372f, 0.749019f);
19 glm::vec1 DAY_ARCADE_SHININESS(1.0f);
20 glm::vec4 DAY_ARCADE_CLEAR(0.28235f, 0.203921f, 0.4588235f, 1.0f);
```

Listing A.2: Matricies definition and object drawing)

A.3 Game Loop

```
void gameLoop()
{
  Shader oceanShader("shaders/ocean.vs", "shaders/ocean.fs");
  Shader lampShader("shaders/lamp.vs", "shaders/lamp.fs");


  while (!glfwWindowShouldClose(window))
  {

    if (isAnimate)
      simulation();

    render(oceanShader, lampShader);
  }
}
```

Listing A.3: Game Loop, containing simulation and render calls

A.4 Keyboard Input

```
void processKeyboardInput(GLFWwindow* window)
{
  if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
    glfwSetWindowShouldClose(window, true);

  //Movement Inputs Commands
  if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
    camera.process_keyboard(FORWARD, delta_time);
  if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
    camera.process_keyboard(BACKWARD, delta_time);
  if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
    camera.process_keyboard(LEFT, delta_time);
  if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
    camera.process_keyboard(RIGHT, delta_time);


  //Scene Setting Toggle
  if (glfwGetKey(window, GLFW_KEY_T) == GLFW_PRESS)
  {
    currentScene++;
    if (currentScene >= SCENE_TOTAL)
    {
      currentScene = 0;
```

```
24      }
25      scene = static_cast<Scene>(currentScene);
26    }
27
28    //Scene Setting Toggle
29    if (glfwGetKey(window, GLFW_KEY_L) == GLFW_PRESS)
30    {
31      isAnimate = !isAnimate;
32
33    }
34
35    //Points Setting Toggle
36    if (glfwGetKey(window, GLFW_KEY_P) == GLFW_PRESS)
37    {
38      isPoints = !isPoints;
39    }
40
41    //Wireframe Setting Toggle
42    if (glfwGetKey(window, GLFW_KEY_O) == GLFW_PRESS)
43    {
44      isWireframe = !isWireframe;
45    }
46
47  }
```

Listing A.4: Keyboard Input handling

A.5 Ocean Vertex Shader

```
1  #version 460 core
2
3  layout (location = 0) in vec4 oceanPos;
4  layout (location = 1) in float oceanHeight;
5  layout (location = 2) in vec2 oceanSlope;
6
7  out VERTEX_SHADER_OUTPUT {
8
9      vec3 normal;
10     vec3 fragPos;
11
12 } vs_Out;
13
14
15 uniform mat4 model;
16 uniform mat4 view;
```

```
17 uniform mat4 projection;
18
19 uniform float heightScale;
20 uniform float chopiness;
21 uniform vec2  size;
22
23 void main()
24 {
25
26     vec3 normal = normalize(cross( vec3(0.0, oceanSlope.y*heightScale, 2.0 / size.
           x), vec3(2.0 / size.y, oceanSlope.x*heightScale, 0.0)));
27
28     vec4 pos = vec4(oceanPos.x, oceanHeight * heightScale, oceanPos.z, 1.0);
29
30     vs_Out.fragPos = vec3(model * pos);
31     vs_Out.normal = mat3(transpose(inverse(model))) * normal;
32
33     gl_Position = projection * view * vec4(vs_Out.fragPos, 1.0);
34 }
```

Listing A.5: Ocean Vertex Shader